ACCELERATING MIXED-LEVEL COVERAGE ARRAY GENERATION

Antoine Veenstra - s1378791

192199978 - Final Project

UNIVERSITY OF TWENTE.

SEPTEMBER 1, 2021

Abstract

The computation of optimum Mixed-level Covering Arrays (MCAs) has various competing algorithms approaching the optimal solution. The IPOG algorithm is a widely used variant and generally produces a small result in a short amount of time. However, generating MCAs of complex SUTs takes a long time, even when using IPOG.

In this report we explain how we accelerate the IPOG algorithm. We employed various techniques to obtain this acceleration, and created prototypes using each technique to evaluate its efficiency. We add an additional representation of the MCA to efficiently detect don't-cares using bitwise operations, which provides a geometric mean speedup of 1.52 on unconstrained SUTs. We divide the work performed by IPOG using multithreading, resulting in a geometric mean speedup of 1.46 for unconstrained SUTs, and 1.18 for constrained SUTs. As an alternative, we create a partial implementation of IPOG using GPGPU. From the results so far, we can conclude that acceleration of the IPOG algorithm can potentially be obtained by creating a GPGPU version of the algorithm. We also discuss the various data structures used, and provide an argumentation as to why said structure was used.

Contents

Chapter 1

Introduction

Testing software is one of the essential steps in software development, as it improves the quality, security and fidelity of software [1]. It is, unfortunately, time-consuming and therefore expensive. Not only the creation of automated tests takes time, but running them also takes time. Making changes to an application will generally require the entire test suite to be run again to guarantee the soundness of these changes. Developers will have to wait for said tests to finish in order to continue development. This is why reducing the size of the test suite can be advantageous to the development speed.

Tests can have certain parameters, such as OS, hardware components, and user input. Each parameter can have a value, such as Windows or Linux for the parameter "OS". The set of parameters and their values defines a System Under Test (SUT). Section 2.1 will further explain what a SUT is. If a fault exists in the software to be tested, then said fault is generally caused by the interaction between a limited amount of values [18, 19, 31]. It is therefore unnecessary to exhaustively test every possible configuration of the SUT. This insight is the basis for Combinatorial Testing (CT). CT is the study of obtaining a test suite with a limited size while keeping a high number of unique interactions between values [31].

The field of CT covers a multitude of problem domains, but this report will focus on the generation of test suites. This test suite is commonly represented by a Mixed-level Covering Array (MCA) [31]. This MCA is a list of tests, each testing a unique configuration of the SUT. An MCA will contain at least one test for each possible interaction to test whether the interaction of these values cause problems. An interaction is the combination of a set of values, the number of values in this interation is equal to the "strength" of an MCA. For example, an MCA would contain all possible interactions between OSs and browsers if the strength is 2. Optionally constraints can be added to limit the possible interactions between certain parameters. For example, not every browser is available on every OS, so tests considering these impossible situations should be left out. Using an MCA, you are guaranteed to find erroneous interactions between values in your SUT. The size of an MCA is far

smaller than an exhaustive test suite, so the time taken during testing will be far shorter when using an MCA. Section 2.2 will further explain the mathematical properties of an MCA.

The generation of test suites is achieved by generating an MCA [15, 31]. Section 2.3 will explore the available algorithms that are known to generate acceptable MCAs. It is important to obtain a balance between getting a smaller solution and keeping the computation times short, during generation of an MCA. Generally, computing the smallest possible MCA is very time consuming, so providing a near optimal solution in a fraction of the time can be more attractive. This is especially the case if the extraneous tests take less time to complete than the time required to find and remove them from the test suite.

In this report we will explore how to accelerate the generation of an MCA. We take an existing MCA generation algorithm, In-Parameter-Order-General (IPOG), and attempt to increase the efficiency using various techniques. This brings us to the following research question:

How do we accelerate the IPOG algorithm?

To answer this questions we create three prototypes of the IPOG algorithm, and additionally create various variations of those prototypes. The prototypes are collectively referred to as LibreCA. We use these prototypes and their variations to evaluate the efficiency of various techniques in an attempt to answer the question.

IPOG provides an MCA in a relatively short amount of time, which makes this algorithm popular. Various implementations have been created [6], competing in resulting MCA size and the time taken to get to said MCA. We will further discuss and explain IPOG, in Chapter 3.

After introducing IPOG, we provide background on the technologies used to attempt acceleration of the algorithm in Chapter 4. We discuss Rust, the programming language used in all prototypes, and how the multithreading prototypes handles communication between threads. We also provide background on General-Purpose computing on Graphics Processing Units (GPGPU), such as the limitations and caveats of this technology. GPGPU is a technique that exploits the previously unused GPU [4].

We thoroughly investigate how to implement the data structures used in our prototypes. By doing this we can obtain the full potential of the various acceleration techniques introduced in the subsequent chapters. We create various benchmarks to provide a fair comparison between the possible data structures. These data structures will be discussed in Chapter 5.

In this report we provide three different approaches to answer our research question, which gives us three subquestions. We answer one subquestion in each of Chapters 6, 7 and 8. Each chapter shows how a technique can be used to accelerate IPOG, and evaluates the techniques using benchmarks.

The first subquestion is: *How do we accelerate the IPOG algorithm using bitwise operations?* We answer this question in Chapter 6. This chapter introduces and evaluates a new technique which uses bitwise operations to increase the overall efficiency of the IPOG algorithm. We created a single-threaded prototype of the IPOG algorithm, and applied this technique to evaluate the resulting speedup.

The second subquestion is: *How do we accelerate the IPOG algorithm using multithreading?* A surefire way of reducing the time required to get an MCA is by using additional resources during the calculations. Multithreading uses the previously idle cores of a CPU to run software in parallel. To answer the subquestion we created a multithreaded prototype of IPOG. In Chapter 7, we discuss how we apply multithreading to accelerate our prototype and compare the multithreading prototype to our single-threaded prototype.

The third subquestion is: *How do we accelerate the IPOG algorithm using GPGPU?* Just like multithreading, GPGPU allows for the use of additional resources to run software in parallel. GPGPU allows for the exploitation of the previously unused GPU to accelerate software. In Chapter 8, we accelerate the IPOG algorithm using GPGPU. This gives us a GPGPU prototype, which we compare to our single-threaded prototype.

Finally, we will conclude this report in Chapter 9. This chapter will also provide various suggestions for future research.

This report introduces two new implementations of the IPOG algorithm, and one partial implementation of the algorithm. We compare various implementation details concerning the data structures used in our prototypes, extending the implementation details described in both [15] and [14]. This report also provides detailed descriptions on how we apply the acceleration techniques mentioned above. In the near future the source of our prototype will be made open source, making this one of the fastest implementations of IPOG available for free.

Chapter 2

Combinatorial Testing

In this chapter we will explain the concepts within the field of Combinatorial Testing (CT) which will be used in the chapters to come. We start with the SUT in Section 2.1, which is the system for which a test is generated or reduced. Section 2.2 is about MCAs, which are essentially mathematical representations of test suites, but with additional properties. These properties are what makes the generation of these test suites challenging, as will be discussed in Section 2.3.

2.1 System Under Test

A test suite tests a System Under Test (SUT), which is represented by three variables: $\langle P, V, \phi \rangle$ [34]. Figure 2.1 provides an example of such a system. P is the set of parameters used in the test suite. In the provided example these parameters are the CPU, Browser, and OS. These parameters may concern any variance in hardware, underlying software, or (user) input. They can have different values, which are provided in V. For example, the values of the parameter CPU are AMD and Intel. One can define a test by choosing a value for each parameter. However, some combinations of values are not possible, so the SUT also has constraints. These are listed in ϕ. One of the constraints listed in the example is that a Mac will not have an AMD CPU.

The SUT listed in Figure 2.1 can be tested using an exhaustive test suite [31], meaning that this test suite will include every possible permutation of values. Table 2.1 shows this exhaustive suite as a two-dimensional array, where each column is a parameter and each cell contains the chosen value. Note that four tests are missing due to their violation of the constraints, as listed in Figure 2.1.

$$
\begin{aligned}
P &= \{\text{CPU}, \text{Browser}, \text{OS}\} \\
V_{\text{CPU}} &= \{\text{AMD}, \text{Intel}\} \\
V_{\text{Browser}} &= \{\text{Firefox}, \text{Chrome}\} \\
V_{\text{OS}} &= \{\text{Linux}, \text{Windows}, \text{Mac}\} \\
\phi &= \begin{cases} \langle V_{\text{CPU}} = \text{AMD} \rangle \implies \neg \langle V_{\text{OS}} = \text{MAC} \rangle \\ \langle V_{\text{Browser}} = \text{Chrome} \rangle \implies \neg \langle V_{\text{OS}} = \text{Linux} \rangle \end{cases}
\end{aligned}
$$

Figure 2.1: The SUT definition based upon the example used in [34].

	V_{CPU}	V_{Browser}	V_{OS}
1	AMD	Firefox	Linux
2	AMD	Firefox	Windows
3	AMD	Chrome	Windows
4	Intel	Firefox	Linux
5	Intel	Firefox	Windows
6	Intel	Firefox	Mac
7	Intel	Chrome	Windows
8	Intel	Chrome	Mac

Table 2.1: The exhaustive test suite for Figure 2.1.

2.2 Mixed-level Covering Array

Within CT, the Mixed-level Covering Array (MCA) represents a specific type of test suite [31]. An MCA is a two-dimensional array of which each row represents a test just like the exhaustive test suite provided in Table 2.1. Each column is a parameter and each cell shows the chosen value for the parameter in the test. In this report we will denote the run size, which is the number of rows, as n. The number of parameters (in P), equal to the number of columns, is denoted as k. So an MCA is an $n \cdot k$ array.

The strength of an MCA is denoted as t. This parameter distinguishes the MCA from the two-dimensional array shown in Table 2.1. It signifies the strength of the t-way interactions, of which the MCA will cover all. An interaction is a set of values for a given Parameter Combination (PC), and a PC is a set of parameters of size t. For example, a 2-way PC for the SUT provided in Figure 2.1 is: $\langle P_{\text{CPU}}, P_{\text{OS}} \rangle$. An example interaction for this PC is: $(\langle V_{\text{CPU}}, \text{AMD} \rangle, \langle V_{\text{OS}}, \text{Mac} \rangle)$.

If $t = 3$ then the MCA will have 3-way coverage, meaning that all possible value interactions for all PCs of size three occur at least once in the test suite.

	V_{CPU}	V_{Browser}	V_{OS}
1	AMD	Chrome	Windows
2	Intel	Firefox	Windows
3	Intel	Chrome	Mac
4	AMD	Firefox	Linux
5	Intel	Firefox	Linux
6	Intel	Firefox	Mac

(a) A minimal 2-way MCA for Figure 2.1.

		V_{Browser}	
		Firefox	Chrome
V_{CPU}	AMD	4	1
	Intel	2, 5, 6	3

(b) CM for the interactions between the values of V_{Browser} and V_{CPU}.

		V_{CPU}	
		AMD	Intel
V_{OS}	Linux	4	5
	Windows	1	2
	Mac	Invalid	3, 6

(c) CM for the interactions between the values of V_{CPU} and V_{OS}.

		V_{Browser}	
		Firefox	Chrome
V_{OS}	Linux	4, 5	Invalid
	Windows	2	1
	Mac	6	3

(d) CM for the interactions between the values of V_{Browser} and V_{OS}.

Figure 2.2: A minimal 2-way MCA and the CMs for the SUT provided in Figure 2.1. The CMs show the numbers of the tests that cover the interaction. The constraints in the SUT explain the missing interactions marked as Invalid in Figures 2.2.c and 2.2.d. The notation of this MCA is $\langle 6, 2, 2^2 3^1 \rangle$.

An example of a 2-way MCA is provided in Figure 2.2.a. This MCA represents a test suite that is not exhaustive.

If the strength is equal to the number of parameters, then the resulting MCA will be equal to the exhaustive test suite of the same SUT. For example, the 3-way MCA of the SUT in Figure 2.1 is exhaustive as there are 3 parameters. This means that the exhaustive test suite provided in Table 2.1 is the 3-way MCA of the given SUT.

With the definition of strength we can also introduce the concept of a minimal MCA. The MCA is called minimal if the smallest possible number of tests is used to cover the test suite, whilst retaining the t-way interaction coverage. In Figure 2.2.a the provided MCA is minimal, as the test suite can be reduced no further without losing an interaction between the values of the parameters. Removing one test from the MCA would result in a test suite that does not cover all possible parameter interaction. It would therefore not be a MCA anymore.

Finally the MCA can optionally have constraints, which are inherited from the SUT. The constraints listed in Figure 2.1 have been applied to the MCA in Figure 2.2.a. The interactions $(\langle V_{\text{CPU}}, \text{AMD} \rangle, \langle V_{\text{OS}}, \text{Mac} \rangle)$, and $(\langle V_{\text{Browser}}, \text{Chrome} \rangle, \langle V_{\text{OS}}, \text{Linux} \rangle)$ are impossible due to these constraints.

Combining the definitions provided above, we define MCAs to be the fol-

lowing: $\langle n, t, v_1 v_2 \cdots v_k \rangle$ [15, 31, 34, 36]. The definition changes slightly between papers, as some will explicitly add the k or the constraints. The variable n is the number of tests in the MCA, t is the strength of the coverage, and the v_x denote the number of values each parameter x has. In Figure 2.2.a the notation of the MCA would be: $\langle 6, 2, 2^2 3^1 \rangle$. Which means 6 tests providing a 2-way coverage over two parameters with two possible values and one parameter with three possible values.

2.2.1 Effectiveness

If the exhaustive test suite detects 100% of the bugs present, then typically the 2-way MCA can detect about 50% to 75% of those same bugs [2, 18]. However, the test suite created using the MCA is smaller than the exhaustive test suite. For instance, the MCA in Figure 2.2.a contains 6 tests, whereas the exhaustive test suite in Table 2.1 contains 8 tests. This difference in size means that running the test suite takes less time, but at the cost of detecting less bugs. So the MCA can be used to provide a preliminary result.

Increasing the strength of the MCA increases the amount of detected bugs, given a large enough SUT. Analyses of empirical data has shown that a 100% bug detection can be obtained with a 4-way to 6-way MCA [18]. These higher strength MCAs are smaller than the exhaustive test suite, so running the MCA test suite will be faster than running the exhaustive test suite. However, in this case the faster test suite does not mean a reduced effectiveness when it comes to the detection of bugs.

There are over 50 tools available when it comes to creation and/or manipulation of MCAs. The most comprehensive list can be found in [6]. These have been used in the real world to improve the test suites in existing projects [12]. For example [33], used CT to design test cases for server concurrent maintenance. By applying CT they improved the reliability next version of their server. This work also includes further application of this technique. Another example is provided in [29], where the authors use CT to generate attack vectors to test the security of web applications. The tool they created helped them find several new critical vulnerabilities in existing software.

2.2.2 Coverage-map (CM)

The data structure used to keep track of the coverage of parameter interactions is called the Coverage-map (CM) [15]. Figures 2.2.b to 2.2.d are the CM for the given MCA in Figure 2.2.a. The number of interactions to keep track of is higher than the amount of tests present, so this data structure plays an important role in algorithms working with MCAs.

The CM is a data structure that will grow significantly in size if the number of parameters increases or if the strength of the interactions increases. If k is the number of parameters, v_x the number of values for parameter x, and t

is the strength of the coverage, then the number of interactions tracked in the CM can be approximated by the following formula:

$$\binom{k}{t} \cdot \left(\frac{\sum v_x}{k}\right)^t$$

This means that the memory usage will increase drastically with a minor increase of the number of parameters or the strength. Tools working with MCAs will have to take into account the CM and find ways to minimise its size when handling big SUTs. The tools will also have to minimise the read and write access or make it as efficient as possible. Each row of an MCA affects $\binom{k}{t}$ interactions, so reasoning about a row will likely require lookup up the coverage status in the CM.

Its size and its integral role when working with MCAs make the CM an important, albeit inconspicuous, part of any tool handling MCAs.

2.3 MCA generation

For a given SUT one can generate an MCA. This MCA will cover all interactions between the chosen parameters. The generation of MCAs is only viable if the software under test runs on random configurations with a limited amount of constraints.

The generation of MCAs is an NP-complete problem [15]. There are various ways to approach the optimal MCA of a given SUT. Each type of algorithms will have different trade-off when it comes to the relation of generation time and MCA size. A general rule is that a slow and efficient algorithm will produce a close to optimal MCA, which is advantageous if the tests themselves take a long time to run. A fast and efficient algorithm will produce a larger test suite, which is advantageous if the tests themselves take a relatively short time to run.

For each real-life scenario the choice of used tool and/or algorithm will depend on the time required to run one test. Other factors like the coverage strength, number of parameters, and parameter levels will all influence the efficiency of the algorithms. Therefore, these need to be considered when choosing the tool and/or algorithm.

Having a deterministic algorithm is advantageous when it comes to generating MCAs. Determinism guarantees constant results and multiple runs of the same tool will provide the exact same results. This also makes the efficiency comparison easier when implementing improved versions of the algorithm.

A SAT solving approach is discussed in Section 2.3.1. Simulated Annealing (SA) is the approach used in Section 2.3.2. We will introduce the greedy family of algorithms in Section 2.3.3.

2.3.1 SAT based algorithms

Generating MCAs using SAT solvers provides a tool that generates small results, but takes a long time to run [34]. It does so by transforming the generation problem to an optimisation problem and tasks a SAT solver to find a solution. Using SAT solvers to solve the problem saves a lot of time when developing tooling. Research in SAT solving is more advanced and enjoys a wider audience than the research in CT, so the optimisations in SAT solving are and will be more advanced than the tools specifically built for CT. Another advantage of the tools using SAT solvers proposed so far is that the results are deterministic. An unnamed tool described in [34] seems to be the leader in this category [6].

2.3.2 Simulated Annealing based algorithms

Just like the SAT based algorithms, the SA algorithms can use existing tooling and techniques to generate MCAs [24]. SA is an algorithm that starts by finding global optimal solutions in the solution domain by random testing. Then it will reduce randomness in order to zero in on the local minimum, which should provide a (near) optimal solution. The advantage of this algorithm is that, just like SAT solving, the amount of SA research available is higher than that of CT research. The biggest disadvantage of SA is the non-determinism of the result. This means that the result will vary per run of the algorithm, so there is no guarantee the provided solution would be the smallest.

The algorithm allows for the tuning of speed depending on the requirements. A slower speed will generally provide improved results when it comes to the MCA size. Higher speeds will do the inverse. CHIP is the leading implementation in this category [24].

2.3.3 Greedy algorithms

A greedy algorithm will make locally optimal choices in the hope that the final outcome is a global maximum. So for each decision the algorithm chooses the best option according to a predefined heuristic. A more complex heuristic could provide better results, but will also take more time to evaluate. This means that greedy algorithms can have better results depending on the speed, but tuning this is not as easy as with SA.

Unlike SAT solvers and SA, greedy algorithms do not have an existing toolset or codebase. This is because the algorithms are designed for the specific problem they are solving. The most popular greedy algorithm for generating MCAs is the In-Parameter-Order-General algorithm, because of its small memory footprint, high performance, and decent results [1, 15]. There are multiple implementations of this algorithm [15, 31, 37]. Some examples of available tools follow. Microsoft Research developed and presumably uses PICT,

which is available in [5]. ACTS is a wellknown tool described in [3]. [15, 32] describe the CAgen tool which, to date, seems to be the fastest tool. These perform faster than the SAT and SA based algorithms [15, 24, 32, 34]. The resulting MCAs are larger, but acceptable considering the speed.

Chapter 3

In-Parameter-Order-General

In this section we will introduce IPOG. First we begin by explaining the overall behaviour and characteristics of this algorithm in Section 3.1. Next, we explore components of IPOG in Sections 3.2 and 3.3. IPOG produces a small MCA in a short amount of time. No other algorithm beats IPOG in both the size and runtime [21, 32]. However, another strong suit of IPOG is the reduced memory requirement, as will be explained in Section 3.4. Section 3.5 will explain how the ordering of the parameters in the SUT influences the resulting MCA. In Section 3.6, we discuss various existing implementations of the IPOG algorithm and their key properties.

3.1 General overview of IPOG

IPOG is a greedy algorithm [15, 21], which means that the entire generation procedure is divided in smaller tasks. Each task is solved by making a locally optimal choice, with the aim of obtaining a globally optimal solution. IPOG fills an MCA step by step, each time choosing the optimal value at the time. This makes the produced MCA the product of locally optimal choices. The MCA is typically not the globally optimal solution, but it is still good considering the performance [3].

IPOG begins with a basic MCA and then adds one parameter at a time. The MCA is extended to cover all the new interactions introduced by the parameter, before continuing to the next parameter. The algorithm does not need to keep track of the coverage of interactions between the previously handled parameters, which reduces the memory usage.

An example execution is provided in Figures 3.1 to 3.3. These figures show the generation of an MCA for the SUT provided in Figure 2.1. IPOG generally runs faster and provides better results if the parameters are sorted on level in decreasing order. This is why the order of the columns in Table 2.1 and Figure 2.2.a is different from the order in Figures 3.1.a, 3.2.a, 3.2.c and 3.3.a. The reason as to why the order is of importance will be discussed in Section 3.5.

	V_{OS}	$V_{Browser}$
1	[Linux]	[Firefox]
2	[Windows]	[Firefox]
3	[Windows]	[Chrome]
4	[Mac]	[Firefox]
5	[Mac]	[Chrome]

(a) The initial MCA generated on Line 2 of Listing 3.1. The filled in values are the product of the values of parameters P_1 and P_2.

		V_{CPU} AMD	Intel
V_{OS}	Linux		
	Windows		
	Mac	Invalid	
$V_{Browser}$	Firefox		
	Chrome		

(b) The CM generated on Line 6 of Listing 3.1. The crossed out value is invalid due to the constraints listed in Figure 2.1.

Figure 3.1: The initial MCA and CM after Line 6 of Listing 3.1. This step is followed by the HE found in Figure 3.2. The 2-way MCA is generated for the SUT provided in Figure 2.1.

The t-way generation begins by creating an MCA for the first t parameters by simply listing all possible interactions between these parameters. The initial MCA for the example generation is shown in Figure 3.1.a. This figure shows the optimal 2-way MCA for the first two parameters. Using this initial MCA we add one parameter at a time. This is done in two steps, namely by HE and Vertical Extension (VE). These steps will be elaborated upon in Sections 3.2 and 3.3 respectively. HE will add the new parameter to each existing row. This extends the rows, which by extension extends the MCA horizontally. VE will add new rows to the MCA, vertically extending the MCA.

Pseudocode of the IPOG algorithm is provided in Listing 3.1, the HE and VE subcomponents are provided in Listings 3.2 and 3.3. Listing 3.1 shows the creation of the initial MCA with the first t columns, on Line 2. Next we add a parameter to the MCA in each iteration of the loop on Line 4 of Listing 3.1. Each iteration consists of initialisation, a HE and optionally a VE. Initialisation, found on Line 6, consists of the creation of the CM for the parameter being added. The resulting CM is provided in Figure 3.1.b. It is empty as no interactions with the new parameter have been covered yet.

3.2 Horizontal Extension

The Horizontal Extension (HE) adds the parameter to be added to each row of the MCA, extending the MCA horizontally. Figures 3.2.a and 3.2.c show the state of the MCA during the HE. The state of the CM can be found in Figures 3.2.b and 3.2.d respectively. For each existing row in the MCA a value for the newly added parameter is assigned. The value assigned should cover the highest number of interactions with the other parameters. In Figure 3.2.a, a new value has been assigned to the first two rows. The new value in the first

```
fn IPOG(parameters, t) -> MCA {
    mca = initial_fill(); // Cross-product of the first t columns

    for p in t..parameters.len() {
        // Create a \(\ac{cm}\) for the newly added parameter
        cm = CoverageMap::new(p);

        // Add the parameter to the existing rows
        horizontal_extension(mca, p, t, cm);

        // Extend the test suite if interactions are missing
        if !cm.is_covered() {
            vertical_extension(mca, p, t, cm);
        }
    }

    return mca;
}
```

Listing 3.1: The IPOG algorithm.

row is “Intel”, and it covers the following interactions:

$$(\langle V_{\text{OS}}, \text{Linux}\rangle, \langle V_{\text{CPU}}, \text{Intel}\rangle) \text{ and } (\langle V_{\text{browser}}, \text{Firefox}\rangle, \langle V_{\text{CPU}}, \text{Intel}\rangle)$$

These interactions have been marked as covered in the CM provided in Figure 3.2.b. The following interactions would be covered if “AMD” was chosen instead:

$$(\langle V_{\text{OS}}, \text{Linux}\rangle, \langle V_{\text{CPU}}, \text{AMD}\rangle) \text{ and } (\langle V_{\text{browser}}, \text{Firefox}\rangle, \langle V_{\text{CPU}}, \text{AMD}\rangle)$$

In the case of two or more options with an equal amount of covered interactions the value is chosen at random. However, this is not the case in all implementations, and breaking this tie using different heuristics will influence the end result [14]. For the purpose of this example we will resolve the tie by choosing a value at random.

The second row in Figure 3.2.a has “AMD” set as value for the new parameter. The following interactions are covered by assigning this value:

$$(\langle V_{\text{OS}}, \text{Windows}\rangle, \langle V_{\text{CPU}}, \text{AMD}\rangle) \text{ and } (\langle V_{\text{browser}}, \text{Firefox}\rangle, \langle V_{\text{CPU}}, \text{AMD}\rangle)$$

The interaction $(\langle V_{\text{OS}}, \text{Windows}\rangle, \langle V_{\text{CPU}}, \text{Intel}\rangle)$ would have been covered if “Intel” had been chosen instead. As we can see in the CM, the interaction $(\langle V_{\text{browser}}, \text{Firefox}\rangle, \langle V_{\text{CPU}}, \text{Intel}\rangle)$ has already been covered, so we do not count this interaction for this row. This means that choosing “AMD” covers two interactions and “Intel” covers only one interaction. The locally optimal choice

	V_{OS}	$V_{Browser}$	V_{CPU}	
1	Linux	Firefox	AMD: 2 Intel: 2	[Intel]
2	Windows	Firefox	AMD: 2 Intel: 1	[AMD]
3	Windows	Chrome	AMD: Intel:	
4	Mac	Firefox	AMD: Intel:	
5	Mac	Chrome	AMD: Intel:	

(a) The MCA after two iterations of the HE are done. It also shows the scores for each possible value.

		V_{CPU} AMD	V_{CPU} Intel
V_{OS}	Linux		[×]
	Windows	[×]	
	Mac	Invalid	
$V_{Browser}$	Firefox	[×]	[×]
	Chrome		

(b) The CM after two iterations of the HE are done.

	V_{OS}	$V_{Browser}$	V_{CPU}	
1	Linux	Firefox	AMD: 2 Intel: 2	Intel
2	Windows	Firefox	AMD: 2 Intel: 1	AMD
3	Windows	Chrome	AMD: 1 Intel: 2	[Intel]
4	Mac	Firefox	~~AMD:~~ Intel: 1	[Intel]
5	Mac	Chrome	~~AMD:~~ Intel: 0	[∗]

(c) The MCA after the HE is done. The "∗" represents a don't-care.

		V_{CPU} AMD	V_{CPU} Intel
V_{OS}	Linux		×
	Windows	×	[×]
	Mac	Invalid	[×]
$V_{Browser}$	Firefox	×	×
	Chrome		[×]

(d) The CM after the HE is done.

Figure 3.2: The MCA and corresponding CM during and after the HE as can be found in Listing 3.2, which is called in Line 9 of Listing 3.1. This step follows after the initialisation found in Figure 3.1, and is followed by VE provided in Figure 3.3. The 2-way MCA is generated for the SUT provided in Figure 2.1.

is made, which is why "AMD" is selected and the CM is updated accordingly. In this report we will refer to the number of interactions to be covered as the score. Figures 3.2.a and 3.2.c both show these scores in the V_{CPU} column.

The value selection for the following rows is shown in Figure 3.2.c. The third row again calculates the score for each value and selects "Intel" as the highest scoring option. For rows four and five, however, the combination of $(\langle V_{OS}, \text{MAC}\rangle, \langle V_{CPU}, \text{AMD}\rangle)$ is not possible, as it is disallowed in the SUT. As a reminder, the SUT of this example can be found in Figure 2.1 on page 6. Rows four and five will only consider "Intel" as a valid option. In row four, "Intel" covers one interaction, which is why this value is chosen. The fifth and last row shows that choosing "Intel" would cover no additional interactions. If none of the options provide coverage of interactions, then the value is not chosen (yet). Instead, a don't-care is filled in, which is represented by a "∗". This value can be chosen at a later stage during the VE, as will be explained in Section 3.3.

Each time a new value is chosen the CM is updated, which results in the

```
1 fn horizontal_extension(mca, p, t, cm) {
2     for row in mca {
3         row[p] = get_best_value();
4         cm.set_covered(row);
5         if cm.is_covered() { return; }
6     }
7 }
```

Listing 3.2: The HE of IPOG.

CM as provided in Figure 3.2.d. There are two more interactions left to be covered. This is handled by the VE step which is discussed in the next section.

The HE function is called on Line 9 of Listing 3.1. Listing 3.2 provides the pseudocode for the HE. The loop from Line 2 till Line 6 will iterate over all rows, as the example from the previous paragraphs illustrate. For each row the best value is selected (Line 3), and the CM is updated accordingly (Line 4). All interactions could be covered during the HE, so Line 5 offers a way to stop the HE to avoid doing useless work.

3.3 Vertical Extension

The example in the previous section shows that the HE does not necessarily cover all interactions. Vertical Extension (VE) takes care of this by iterating over each uncovered interaction and adding it to the MCA. There are two ways of adding an interaction. VE first tries to fit the interaction into an existing row. This will be discussed in Section 3.3.2. If this does not succeed, then VE add a new row to the MCA. Section 3.3.1 will provide an example of this. The VE extends the MCA vertically by adding rows, which is the reason for the name of this extension.

Figure 3.3 provides the pseudocode of VE. The loop from Lines 2 to 11 iterates over all uncovered interactions. Lines 3 to 9 try to add the interaction to an existing row. Line 10 will add the interaction as a new row to the MCA. The two methods allow VE to cover all remaining interactions, after which IPOG will continue to the next parameter. If there is no more parameter to be added, then the algorithm has finished.

3.3.1 New row

If VE can not fit the interaction into an existing row, then a new row is created with the interaction filled in, and the rest of the values set to don't-cares. HE leaves two interactions uncovered in the example provided in Figure 3.2:

$$(\langle V_{\text{OS}}, \text{Linux}\rangle, \langle V_{\text{CPU}}, \text{AMD}\rangle) \text{ and } (\langle V_{\text{browser}}, \text{Chrome}\rangle, \langle V_{\text{CPU}}, \text{AMD}\rangle)$$

```
fn vertical_extension(mca, cm) {
    'interactions: for interaction in cm.uncovered {
        for row in mca {
            if interaction_fits(row, interaction) {
                row.set(interaction);
                cm.set_covered(row);
                continue 'interactions;
            }
        }
        mca.append(row with interaction); // Remaining values set to "*"
    }
}
```

Listing 3.3: The VE of IPOG.

	V_{OS}	$V_{Browser}$	V_{CPU}
1	Linux	Firefox	Intel
2	Windows	Firefox	AMD
3	Windows	Chrome	Intel
4	Mac	Firefox	Intel
5	Mac	Chrome	$*$
6	[Linux]	[$*$]	[AMD]
7	[$*$]	[Chrome]	[AMD]

(a) The MCA after the VE. Each $*$ represents a don't-care.

		V_{CPU} AMD	V_{CPU} Intel
V_{OS}	Linux	[×]	×
	Windows	×	×
	Mac	Invalid	×
$V_{Browser}$	Firefox	×	×
	Chrome	[×]	×

(b) The CM after the VE.

Figure 3.3: The MCA and corresponding CM after the VE. This step is executed after the HE found in Figure 3.2. The 2-way MCA is generated for the SUT provided in Figure 2.1.

These interactions do not fit into the rows 1 to 4, as these have no don't-cares that can be changed. The fifth row has MAC as the V_{OS}, and the constraints listed in Figure 2.1 prevent the combination of MAC and AMD. This means that the uncovered interaction can not be inserted into any existing row.

Row 6 of Figure 3.3.a shows the first inserted interaction. The next interaction does not fit into the newly created row, as the combination of Linux and Chrome is disallowed by the constraints. This interaction is added to the MCA as a new row.

These two additional rows cover the remaining interactions, so the resulting MCA contains all interactions between each pair of parameters.

3.3.2 Existing row

An interaction might fit into an existing row by setting these don't-cares to specific values. To illustrate this we extend the SUT given in Figure 2.1 with an

$$
\begin{aligned}
P &= \{\text{CPU}, \text{Browser}, \text{OS}\} \\
V_{\text{CPU}} &= \{\text{AMD}, \text{Intel}\} \\
V_{\text{Browser}} &= \{\text{Firefox}, \text{Chrome}\} \\
V_{\text{OS}} &= \{\text{Linux}, \text{Windows}, \text{Mac}\} \\
V_{\text{IPv}} &= \{\text{IPv4}, \text{IPv6}\} \\
\phi &= \begin{cases} \langle V_{\text{CPU}} = \text{AMD} \rangle \implies \neg \langle V_{\text{OS}} = \text{MAC} \rangle \\ \langle V_{\text{Browser}} = \text{Chrome} \rangle \implies \neg \langle V_{\text{OS}} = \text{Linux} \rangle \end{cases}
\end{aligned}
$$

Figure 3.4: The SUT provided in Figure 2.1, but with one extra parameter. V_{IPv} defines the IP version used by the system.

extra parameter for the IP version. The resulting SUT is provided in Figure 3.4. Using this extra parameter we continue the example provided in Listing 3.3, and do another iteration of the IPOG algorithm. How HE works is already explained in Section 3.2, so we will skip that and provide you with the state of the MCA after HE in Figure 3.5.a. During the HE the CM is updated accordingly, the result of which is shown in Figure 3.5.b.

The last interaction to cover is as follows:

$$(\langle V_{\text{OS}}, \text{Windows} \rangle, \langle V_{\text{IPv}}, \text{IPv4} \rangle)$$

We will try to fit this interaction into an existing row. Rows 1 to 4 do not have any don't-cares in them, so we can not fit our interaction in them. Rows 5 and 6 both have another V_{OS} than the one in the interaction, so the interaction also does not fit into these rows. Finally, we arrive at row 7. The V_{OS} is set to don't-care and the V_{IPv} is set to IPv4 like the interaction. By changing the V_{OS} from don't-care to Windows, we are able to cover the remaining interaction without adding more rows to the MCA. The resulting MCA is shown in Figure 3.5.c.

Listing 3.3 shows the pseudocode of the VE, of which Lines 3 to 9 will try to add the interaction to an existing row. The loop from Lines 3 to 9 iterates over the MCA. This also includes recently added rows. Then, Line 4 will check if the interaction fits in the row. If it does not, then the loop will continue to the next row. If the interaction fits in the row, then the interaction is filled in as shown on Line 5. Changing values of a row might also cause it to cover additional interactions, so on Line 6 these additional interactions are set accordingly in the CM.

3.4 Coverage-map size

An advantage of IPOG is the reduced size of the CM. This section will explain the difference in size of the CM during IPOG and alternative algorithms. We

	V_{OS}	V_{Browser}	V_{CPU}	V_{IPv}
1	Linux	Firefox	Intel	IPv4
2	Windows	Firefox	AMD	IPv6
3	Windows	Chrome	Intel	IPv6
4	Mac	Firefox	Intel	IPv6
5	Mac	Chrome	$*$	IPv4
6	Linux	$*$	AMD	IPv6
7	$*$	Chrome	AMD	IPv4

(a) The MCA after the second HE. Each $*$ represents a don't-care.

		V_{IPv} IPv4	IPv6
V_{OS}	Linux	×	×
	Windows		×
	Mac	×	×
V_{Browser}	Firefox	×	×
	Chrome	×	×
V_{CPU}	AMD	×	×
	Intel	×	×

(b) The CM after the second HE.

	V_{OS}	V_{Browser}	V_{CPU}	V_{IPv}
1	Linux	Firefox	Intel	IPv4
2	Windows	Firefox	AMD	IPv6
3	Windows	Chrome	Intel	IPv6
4	Mac	Firefox	Intel	IPv6
5	Mac	Chrome	$*$	IPv4
6	Linux	$*$	AMD	IPv6
7	[Windows]	Chrome	AMD	IPv4

(c) The MCA after the second VE. Each $*$ represents a don't-care.

Figure 3.5: The MCA and corresponding CM before the second VE, and the MCA after second VE. The 2-way MCA is generated for the extended SUT provided in Figure 3.4.

begin by calculating the size of a CM for all interactions.

To calculate the size of the CM capable of keeping track of all interactions between parameters we need two things. First, we need a list with all the Parameter Combinations (PCs). A PC is an unordered subset of parameters of size t, where t is the strength of the coverage of the t-way MCA. Every combination of parameters of size t is present in the list of PCs. Each PC has a list of possible values assigned to each parameter. This is the list of interactions, given a PC. The size of this list is the second number we need to calculate the size of the CM.

For each PC we calculate the number of interactions as follows:

$$\prod_{\text{parameter}}^{\text{PC}} |V_{\text{parameter}}|$$

Where $|V_{\text{parameter}}|$ is the number of values for said parameter. If we sum the number of interactions for each PC then we will find the size of the CM:

$$\sum_{\text{PC}}^{\text{PC list}} \left(\prod_{\text{parameter}}^{\text{PC}} |V_{\text{parameter}}| \right)$$

For example, the 2-way MCA of the SUT provided in Figure 2.1 has a CM of the following size:

$$\begin{aligned}|\text{CM}| &= |V_{\text{CPU}}| \cdot |V_{\text{Browser}}| + |V_{\text{CPU}}| \cdot |V_{\text{OS}}| + |V_{\text{Browser}}| \cdot |V_{\text{OS}}| \\ &= 2 \cdot 2 + 2 \cdot 3 + 2 \cdot 3 \\ &= 16\end{aligned}$$

So the total number of interactions to be covered by the 2-way MCA is 16, which will also be the size of the CM.

However, the size of the CM used by IPOG in the example illustrated by Figures 3.1.b, 3.2.b, 3.2.d and 3.3.b is 10. This is due to the fact that IPOG only needs to check the interactions concerning the parameter to be added. The interactions between $|V_{\text{Browser}}|$ and $|V_{\text{OS}}|$ do not need to be checked as these do not the concern the parameter to be added.

The reduction of the size will be more apparent with a bigger SUT. For example, take an SUT with 32 parameters, each with 8 values, and a strength of 6. Then the number of interactions per PC is always the same, as the number of values is constant. So each PC has the following number of interactions:

$$\prod_{\text{parameter}}^{\text{PC}} |V_{\text{parameter}}| = \prod^{\text{strength}} 8 = 8^6 = 262\,144$$

To calculate the number of total number of PCs we need to find out how many times we can choose 6 unique parameters out of the 32 available where the order does not matter. This is equal to the following:

$$\binom{|\text{parameters}|}{t} = \binom{32}{6} = 906\,192$$

So the size of a CM for all interactions is $262\,144 \cdot 906\,192 = 237\,552\,795\,648$, which is about 30 GB.

IPOG will not need a CM for all interactions, but at most the interactions concerning one parameter and all the parameters previously added. This means that every PC will have one parameter set with the parameter to be added. The remaining spots of the PC can be chosen out of the remaining parameters. This means that the number of parameters to choose from is 31, not 32. It also means that the number of parameters to be chosen for each PC is 5, not 6. We can calculate the number of PCs used during IPOG as follows:

$$\binom{|\text{parameters}| - 1}{t - 1} = \binom{31}{2} = 169\,911$$

The maximum size of the CM during IPOG is $262\,144 \cdot 169\,911 = 44\,541\,149\,184$, which is about 6 GB.

The size of the CM during IPOG is over 5 times smaller than the maximum size. To be precise the reduction is:

$$\begin{aligned}\text{reduction} &= \frac{\binom{|\text{parameters}|}{t} \cdot \prod_{\text{parameter}}^{\text{PC}} |V_{\text{parameter}}|}{\binom{|\text{parameters}|-1}{t-1} \cdot \prod_{\text{parameter}}^{\text{PC}} |V_{\text{parameter}}|} \\ &= \frac{\binom{|\text{parameters}|}{t}}{\binom{|\text{parameters}|-1}{t-1}} = \frac{|\text{parameters}|}{t} \\ &= \frac{32}{6} = 5.\overline{3}\end{aligned}$$

This day and age computers have 16 GB to 32 GB of RAM, so a CM of 30 GB will not fit next to the MCA and the OS. However, a CM of 6 GB will fit into memory with plenty of room to spare. Having the CM in memory allows for faster read and writes, improving overall performance of IPOG.

3.5 Ordering of the variables

Every change in the detailed implementation of IPOG can influence the result [14]. The ordering of the parameters will have effect on the entire run of the algorithm. This section will explain why the parameters are sorted based on descending cardinality. Figure 3.5.c shows an MCA with its parameters sorted based on descending cardinality. Figure 3.6 shows an MCA using the reverse sorting for its parameters. Note that the two MCAs are completely different, but both are valid MCAs.

The first t parameters will be used to create the initial MCA. As shown in Figure 3.1, this MCA is an exhaustive list of all possible interactions between

	V_{CPU}	V_{Browser}	V_{IPv}	V_{OS}
8	AMD	Firefox	IPv4	Linux
9	AMD	Chrome	IPv6	Windows
10	Intel	Firefox	IPv6	Mac
11	Intel	Chrome	IPv4	Windows
12	Intel	Firefox	IPv6	Linux
13	$*$	Firefox	$*$	Windows
14	Intel	Chrome	IPv4	Mac

Figure 3.6: This is the 2-way MCA that would be generated for the SUT provided in Figure 3.4, if the ordering of the parameters was based on ascending cardinality. Each $*$ represents a don't-care.

$\|P\|$	3	2	2	2
1	0	0	0	0
2	1	0	0	1
3	2	0	0	0
4	0	1	0	1
5	1	1	0	0
6	2	1	0	1
7	0	0	1	1
8	1	0	1	0
9	2	0	1	1
10	0	1	1	0
11	1	1	1	1
12	2	1	1	0

(a) Sorting based on descending cardinality.

$\|P\|$	2	2	2	3
1	0	0	0	0
2	1	0	0	1
3	0	1	0	2
4	1	1	0	0
5	0	0	1	1
6	1	0	1	2
7	0	1	1	0
8	1	1	1	1
9	0	0	1	2
10	0	1	0	1
11	1	0	1	0
12	1	1	0	2
13	*	0	0	2
14	*	1	1	2

(b) Sorting based on ascending cardinality.

Figure 3.7: These two MCAs both provide 3-way coverage of an unconstrained SUT with parameter levels $3^1 2^3$. Both are creating using the IPOG algorithm, but they differ in ordering of parameters. Each $*$ represents a don't-care.

the parameters. In practice, generating this initial MCA will take less time than the subsequent HEs and VEs. From this we can conclude that speeding up the extensions will have a bigger overall impact than reducing the time spend on the initial MCA.

In order to reduce the time spend on the extensions, we can decrease the cardinality of the parameters added during those extensions. Lowering the cardinality will reduce the number of interactions that have to be considered during the extension. In turn, this decreases the time spend on finding the number of to be covered interactions during HE and it will reduce the number of interactions added during VE.

We provide an example to illustrate how the ordering of parameters can influence the size of the resulting MCA. Figure 3.7 shows two MCAs, both for the same SUT. The SUT is unconstrained, and cardinality of the levels is $3^1 2^3$. The difference between the two is the ordering of the parameters. Figure 3.7.a shows the MCA where the parameters have been sorted based on descending cardinality. Figure 3.7.b shows the MCA where the parameters have been sorted based on ascending cardinality. The two MCAs also differ in number of rows, the one with descending cardinality based sorting sporting two fewer rows. This is the result of IPOG choosing the incorrect values due to its greedy nature. The chosen value might be the best fit for the row, but not for the overall MCA. Values with high levels will provide more opportunities for incorrect choices, so handling the higher levels in the initial MCA reduces the chances

Feature	ACTS	CAgen	LIBRECA
Maximum strength	unlimited	8	12
Constraints supported	Yes	Yes	Yes
Language	Java	Rust	Rust
Distribution	Binary	Binary and webapp	Binary
User interface	GUI and CLI	Webapp and CLI	CLI
Open source	No	No	Soon™

Figure 3.8: This table shows the various features supported by ACTS, CAgen, and LIBRECA [3, 32].

of incorrect choices later on. Sorting the parameters based on descending cardinality produces smaller MCAs on average [14].

Sorting the parameters based on descending cardinality will increase the time spend on generating the initial MCA, but reduce the time spend on both HE and VE. It will also reduce the size of MCAs on average. There are no known disadvantages to this ordering [3, 15, 32]. This is why most, if not all, implementations of IPOG will order the parameters as such.

3.6 Alternative versions

IPOG was first introduced as In-Parameter-Order (IPO) in [20, 38]. IPO can only create MCAs of which the parameters have a fixed number of values. An MCA (Mixed-level Covering Array) where the parameters are not of mixed levels (cardinality) is simply called a Covering Array (CA). The advantage of CAs is that the generation is easier, as the algorithm does not need to consider the cardinality of each parameter. IPOG then extended this algorithm to also support SUTs with mixed levels in [21]. Since then multiple variants of IPOG have been created [6].

In this report we compare our prototypes to two other implementations in Section 7.2. These are CAgen and ACTS. CAgen is, to date, the most efficient implementation of IPOG available for public use. In [14, 15, 32] the authors of CAgen explore multiple implementation techniques of IPOG. The prototypes described in this report follow the design techniques described.

In Figure 3.8, we provide a comparison between ACTS, CAgen, and LIBRECA. Binaries of both ACTS and CAgen can be requested from their respective authors. CAgen alternatively provides a webapp to their tool. This webapp is presumably created by compiling their Rust code to webassembly. This allows for browsers to run the code at speeds comparable to the equivalent binary version. ACTS sports a GUI with more features than the webapp of CAgen. This makes ACTS the tool to use for advanced tasks, whereas CAgen provides easier access for less complex tasks. Section 4.1 will provide details on how we benchmark these tools.

Chapter 4

Technologies used

In this chapter we discuss the technologies used to implement the prototypes. Throughout Chapters 5 to 8, we use benchmarks to evaluate the various design decisions and implementation techniques. We provide some background on benchmarks in Section 4.1. All of our prototypes precompile the algorithm for a limited number of MCA strengths. In Section 4.2, we discuss what this entails. In our multithreading prototype we require communication between the threads. We discuss how various libraries and Rust built-in methods compare in Section 4.3. Our GPGPU prototype uses Rust, but also CUDA. We provide background on CUDA and GPGPU in Section 4.4.

Our prototypes all use Rust as their primary programming language. We choose this language because of our positive experiences in the past when using this language. The performance is comparable to C and C++ [30]. The ownership model enforces memory safeness, reducing the number of memory management related bugs. One main disadvantage is that the language is relatively young, and some of the language features used during the implementation of our prototypes are not stable yet.

Our single and multithreaded prototypes, provide support for SUTs with constraints. We use SAT solvers to resolve these constraints during generation. The prototypes can use the Z3, MiniSAT, and Glucose solvers to perform this task. Preliminary results show that the MiniSAT solver outperforms the other solvers. However, future research could add support for additional solvers and compare the performance of these.

4.1 Benchmarks

In our report we list two types of benchmarks. One type runs a small code fragment a number of times and then gives the average runtime of this fragment. These benchmarks make it easier to compare the performance of code snippets. The other type takes the runtime of the generation of an MCA. We use the SUTs listed in Figure A.1 (Page 85) and generate 2-way to 12-way MCAs

wherever applicable. These benchmarks allow for the evaluation of design decisions concerning larger code fragments.

The results of any benchmark are specific to the device running them, the Rust compiler version, and the compilation settings. Changes in any of these will inevitably change the results, which could lead to different conclusions when deciding how to implement the prototype. This is unfortunate, but unavoidable.

All benchmarks listed in Chapters 5 to 8 are run on a machine with a *Intel Core i7-6700 CPU @ 3.40GHz* with 64GiB RAM (4x DDR4 Synchronous 2133MHz) running Ubuntu 18.04.4 LTS (Linux 4.15.0-91-generic). We implement IPOG using Rust 1.42 (stable).

4.2 Precompiled constants

The authors of [15] describe an implementation detail which allows for a significant performance boost. They compile their implementation for each supported strength, which means that generation of a 2-way MCA will use other code than the generation of a 3-way MCA. This allows for the unrolling of loops, removing various bounds checks, facilitating memory management, and more. In our prototypes we also apply this technique.

In Section 5.2, Figure 5.8 shows how precompiling can improve performance. However, there are two disadvantages: The compilation of our prototypes is taking longer than before, and the prototypes will only work for a limited number of strengths. Both of these disadvantages have no direct impact on the performance of the program. That is why our prototypes also use this technique.

Our prototypes have a unique code path for each supported strength, which are 2-way to 12-way MCAs. The correct path is selected at the start of the run, after which the code can be optimised for the specific strength. This means that PCs and interactions have known sizes, allowing us to use `array`s for these data types. Operation using these data types can also be optimised, as loops iterating over them will have known sizes which allows for loop unrolling.

4.3 Thread communication

In our multithreaded prototype we require some sort of communication between threads to orchestrate their efforts. In Figure 4.1 we show various options available, each with their efficiency visualised. We have left out all options that can not realistically be used in our prototype. This gives us two options in the built-in library of Rust, namely the barrier and the channel. We also evaluated the options provided by the Crossbeam library that can be used

Communication type	**Time** (ns)
Built-in barrier	12350
Built-in channel	505
Crossbeam channel bounded	224
Crossbeam channel unbounded	201

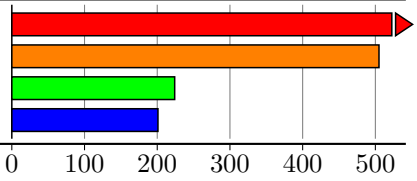


Figure 4.1: This table provides the results of the benchmarks measuring the speed of various types of communication between threads. The first two rows show two built-in communication types. These are provided in the standard library of Rust. The last two rows show the performance of the channel implementation of the Crossbeam library.

in our prototype. This gives us the channel in its bounded and unbounded flavors.

The benchmark starts multiple threads that busy themselves with tasks related to IPOG. After each task the thread communicates its progress. We measure the time it takes for all threads to finish, which includes the receiving of all communications. We compare this time to a benchmark that does not communicate. The difference is the result provided in Figure 4.1.

The benchmark suggests that the Crossbeam library provides the most efficient way to communicate between threads. However, even this implementation possibly takes more time than the score calculation during HE or the search for compatible rows during VE. Any multithreaded implementation should reduce the number of messages passed between threads. We explain how we achieve this in our prototype in Section 7.2. If the overhead created by the communication is too significant then the prototype should run the single-threaded code instead. Our implementation will use the multithreaded implementation only if it deems the amount of work to be sufficient. Currently the heuristic to determine when to switch is the amount of PCs. Future research could compare other heuristics and their impact on the overall performance of the prototype.

4.4 GPGPU

In our General-Purpose computing on Graphics Processing Units (GPGPU) prototype we use CUDA to run code on the GPU. More information about CUDA is provided in Section 4.4.2. We describe how we apply GPGPU in Chapter 8. The rest of this section will provide a background on GPGPU.

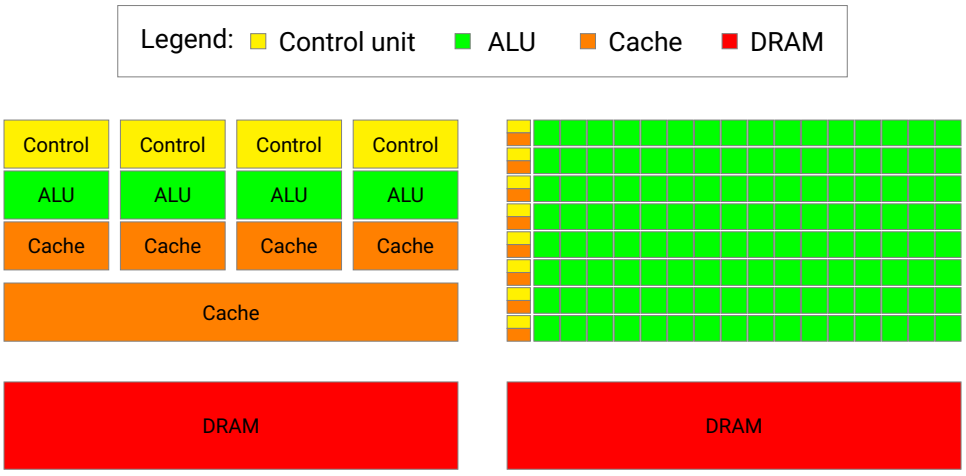
GPGPU is the use of Graphics Processing Units (GPUs) for general-purpose programming [4]. A GPU is hardware responsible for the creation and manipulation of imagery intended to be shown on the screen of a device.

General-purpose programming is the computation of equations belonging to any domain. So, for instance, the computation of flow simulations can be done on GPUs [10]. The main advantage of GPGPU is the utilisation of previously unused resources that are commonly present in devices. Most laptops and desktops have the capability of showing a graphical interface, so most will have a GPU. Even low performance components can speed up the calculations, because of the increase in overall processing capacity.

Most of the strengths and caveats of GPGPU have their roots in the architecture of the GPUs when compared with CPUs [13, 23]. The specifics of this architecture will be discussed in Section 4.4.1. Section 4.4.2 will be about OpenCL and CUDA respectively, which are two popular GPGPU frameworks. These sections will also provide code examples showing the usage of the GPU.

4.4.1 GPU architecture

Understanding the GPU architecture is easier with some knowledge of the CPU architecture. The differences between the two are what makes GPGPU difficult to exploit, but they also provide reason as to why to use GPGPU. This is why we will highlight differences between the two architectures.



(a) Quad-core CPU (b) GPU

Figure 4.2: Simplified structure visualisation of a CPU and a GPU [11, 27].

Figure 4.2 provides a simplified representation of both the CPU and GPU in order to visualise the differences between the two. Figure 4.2.a shows a quad-core CPU, which is composed of four cores. Each core has its control unit, an Arithmetic Logic Unit (ALU), and a cache. A typical CPU has multiple layers of caching in order to limit the memory lookups, which are time-consuming.

Figure 4.2.b shows a GPU. It consists of an ALU for each hardware thread, and per group of threads there is a control unit and cache. Transformations performed on graphics are generally equivalent for each pixel and/or vector. This means that the cores of a GPU will run the same command, but use different data. GPU manufacturers have chosen to exploit this property by assigning only one control unit for multiple ALUs [23]. Removing control units from cores leaves you with hardware threads.

Components of the GPU are generally smaller and less complex than their counterparts in the CPU. Each core of a CPU will have a control unit, an ALU, and a cache [27]. CPUs are designed to process a set of instructions in the least amount of time. Just like the GPU, the hardware is optimised for its most common tasks. However, there are no given rules or limitations to the nature of the problems to be solved by a CPU. Therefore, hardware optimisations are focused accelerating basic instructions like branching and accessing memory [16]. GPUs have a more focused task descriptions, namely the creation and manipulation of imagery. The control unit is simpler, as typical CPU features, such as branch predictions and cache prefetching, are not included. Branch prediction is a technique which tries to predict which branch a program will take, so that the next instructions can be loaded and prepared in advance. The control units do not include branch prediction, because this is too complex for a larger group of hardware threads [11]. Cache prefetching is a technique where memory data will be fetched to the cache before it is needed. However, cache prefetching without a large enough cache is counterproductive, that is why this too is not included in the control unit of a typical GPU [22]. ALUs are focused on solving graphics specific problems, so floating-point and vector transformations are typically supported. Other instructions are rarely executed on a GPU, so their circuitry is less advanced and sometimes limited. Higher levels of cache shared among cores is common in a CPU. The cache of a GPU is smaller to cut cost, and because any amount of cache would be unable to store the entirety of the input data anyway. So, all the components of the GPU have been adapted to its primary task, and thus differ from the components present in a CPU.

Figure 4.2.a shows a CPU with multiple cores, but the original CPU architecture calls for only one core. The addition of more cores to the CPU is not a direct result of the need for faster execution [4]. Speed of a CPU is physically limited, namely the cooling mechanisms not being able to dissipate the heat adequately. The heat generated by CPUs is quadratically related to the speed. However, doubling the number of cores only doubles the heat generated. So in order to increase the overall performance of a CPU it is easier to add more cores than speeding it up.

This heating boundary is completely circumvented by the GPU. The GPU focuses on a higher amount of cores, and not on the speed of each core. This high amount of cores allows for the concurrent processing of data. Graphics are large amounts of date, so the GPU will profit from this high amount of

cores when creating and manipulating imagery.

Using the GPU to perform calculations not related to graphics is possible. However, the architecture as discussed above will induce limitations on the way said calculations can be performed compared to traditional CPU based calculations. There are also features exclusive to the GPU that can be exploited for increased performance if applied correctly. For example, the high amount of threads of a GPU can concurrently transform large amounts of data, something a CPU is incapable of [4, 13, 23]. Fortunately there exist frameworks which facilitate development of programs run on the GPU. The next section will discuss two of them.

4.4.2 GPGPU frameworks

CUDA and OpenCL are the two frameworks commonly used for GPGPU programming. Both of the frameworks share the way they execute code on the GPU [25, 27]. The CPU acts as the host and instructs the GPU what to do. A GPU does not have access to most peripherals, so sporadic communication between the CPU and GPU is common. Most communication between the two devices is done by reading and writing to the global memory of the GPU.

The CPU instructs the GPU to run a so-called kernel. In both frameworks the kernel is compiled at runtime, so that the program can run on different devices without the need for the distribution of binaries for each existing GPU. The kernels are written in a C-like programming languages. These languages have additional instructions for the GPU specific features.

The program on the host is not restricted to a certain programming language, so choosing your favorite language should be possible. However, the frameworks have better integration with a select group of languages. C, C++, and related languages are favored in both frameworks. The host language used in the examples below is Rust, as our prototypes are written in this language.

OpenCL

OpenCL is a framework supported by all main GPU manufacturers [25]. It is maintained by Khronos, but each GPU vendor has to implement the support themselves. The most important advantage of this is the widespread support for this framework on devices [17]. When designing an application that should run on various devices, then this is the framework to choose. OpenCL not only supports GPUs, but also CPUs. This means that programs using OpenCL will run on every device.

However, the widespread support is also its biggest disadvantage, as companies have no incentive to develop decent support for the framework. It is financially more attractive to make an alternative framework that supports only their products in order to prevent customers from switching to

their competitors. This why for example Nvidia created CUDA, Microsoft created DirectCompute, Apple created Metal, and Google created RenderScript. Support for OpenCL will not have a priority for these companies.

```
__kernel void add(__constant float *x, __constant float *y,
                  __global float *out, uint count) {
    uint index = get_global_id(0);
    if (index < count) out[index] = x[index] + y[index];
}
```

Listing 4.1: Example of an OpenCL program.

```
use ocl::{error::Result as OclResult, ProQue};

const N: usize = 10000; const XS: f32 = 30.0; const YS: f32 = 7.0;
const SRC: &str = include_str!("add.cl");

fn main() -> OclResult<()> {
    // Init OpenCL, get a device, compile a kernel, and set dimensions.
    let pro_que = ProQue::builder().src(SRC).dims(N).build()?;

    // Create buffers as big as the specified dimension. Fill x and y.
    let x = pro_que.buffer_builder().fill_val(XS).build()?;
    let y = pro_que.buffer_builder().fill_val(YS).build()?;
    let out = pro_que.buffer_builder().build()?;

    unsafe { pro_que.kernel_builder("add").arg(x).arg(y).arg(&out).arg(N)
                    .build()?.enq()?; } // Load and enqueue the kernel.
    pro_que.flush()?; // Wait for kernel to finish
    let mut value = vec![0f32]; // Prepare location for result
    out.read(&mut value).offset(N - 2).enq()?; // copy result
    println!("Sum is {}", value[0]);
    Ok(())
}
```

Listing 4.2: Example of using OpenCL in Rust.

Listings 4.1 and 4.2 provide an example program using OpenCL. It adds all the numbers of two buffers and writes the result to another buffer. Listing 4.1 provides the kernel written in a C like language based on C99 [25]. The kernel `add` is called by the host code and will run in parallel on all hardware threads of the GPU. On Line 1 the definition of the function is given. Line 3 shows the retrieval of the unique id of the thread as the index, and Line 4 shows the assignment of the sum at this index.

Listing 4.2 shows the host code of the example. It is written in Rust, using a package called *"ocl"* [1]. This package abstracts most of the interactions with the OpenCL driver, while keeping the possibility of falling back on the raw driver. This structure makes the development easy, while allowing for performance related tweaks. On Line 8 of Listing 4.1 the GPU is initialised, and the kernel is loaded. The kernel, shown in Listing 4.1, is compiled only if a cached version is not available. Meaning that the first run of a program will be slower due to the compilation, but the next iteration will not have the same performance hit caused by the compilation. Compilation to binaries is not possible, as the compilation is done by the device specific driver and the compilation result will differ per device. Lines 11 to 12 load values into the GPU memory, and Line 13 reserves GPU memory for the result. Line 15 will launch the kernel on the GPU. The memory actions and the launch will be done asynchronously, but in order. This means that before the GPU code is launched, all the memory actions will have taken place. On Line 17 the host waits for the asynchronous calls to finish. Line 19 copies one of the results from the GPU memory to the host memory. This is then printed on Line 20. Run this program with the following command: `cargo run`. The code can be cross compiled for most platforms, and for each the Rust compiler can provide a single binary file. This makes the distribution of the program easy.

CUDA

CUDA is the GPGPU framework developed by NVIDIA, which works exclusively on their products [26]. An important advantage of this framework is the high level of support provided. Another advantage is that the framework can make all features of NVIDIA GPUs available to the programmer. NVIDIA has some proprietary hardware features that can improve the performance of GPGPU programs [23]. The biggest disadvantage is that any program using CUDA is useless when run on hardware without a NVIDIA GPU. This severely limits the portability of any project using this framework.

Just like Listings 4.1 and 4.2, Listings 4.3 and 4.4 provide an example program, but this time using CUDA. It also adds all the numbers of two buffers and writes the result to another buffer. Listing 4.3 provides the kernel written in a `C++` like language specific to CUDA [27]. The kernel `add` is called by the host code and will run in parallel on all hardware threads of the GPU. The structure of this example is the same as the one for OpenCL in Listing 4.1. On Line 2 the definition of the function is given. Line 3 shows the retrieval of the unique id of the thread as the index, and Line 4 shows the assignment of the sum at this index.

Listing 4.4 shows the host code of the example. It is also written in Rust, using a package called *"rustacuda"* [2]. This package provides a few abstrac-

[1] Check out crates.io/crates/ocl for more information about *"ocl"*.

[2] Check out crates.io/crates/rustacuda for more information about *"rustacuda"*.

```
extern "C" __global__
void add(float *x, float *y, float *out, int count) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < count) out[index] = x[index] + y[index];
}
```

Listing 4.3: Example of a CUDA program.

```
use std::{cmp::*, ffi::*, mem::*}; use cuda_sys::cuda::*;
use rustacuda::{*, device::DeviceAttribute::*, prelude::*};

const KERNEL: &[u8] = include_bytes!("add.ptx"); // Run: nvcc add.cu
↪  --ptx
const N: usize = 10000; const XS: f32 = 30.0; const YS: f32 = 7.0;

fn main() -> Result<(), Box<dyn std::error::Error>> { unsafe {
    rustacuda::init(CudaFlags::empty())?; // Initialize the CUDA API.
    let dev = Device::get_device(0)?; // Get the first device.
    let block_size = min(dev.get_attribute(MaxBlockDimX)? as u32,
                         N as u32);
    let grid_size = N as u32 / block_size + 1;
    let _c = Context::create_and_push(
        ContextFlags::MAP_HOST | ContextFlags::SCHED_AUTO, dev)?;
    let stream = Stream::new(StreamFlags::NON_BLOCKING, None)?;
    let module_data = CString::new(KERNEL)?; // Load kernel
    let module = Module::load_from_string(&module_data)?;

    // Allocate space on the device and copy numbers to it.
    let mut x = DeviceBuffer::<f32>::uninitialized(N)?.as_device_ptr();
    cuMemsetD32_v2(x.as_raw_mut() as u64, XS.to_bits(), N);
    let mut y = DeviceBuffer::<f32>::uninitialized(N)?.as_device_ptr();
    cuMemsetD32_v2(y.as_raw_mut() as u64, YS.to_bits(), N);
    let out   = DeviceBuffer::<f32>::uninitialized(N)?.as_device_ptr();

    launch!( // Enqueue the kernel
        module.add<<<grid_size, block_size, 0, stream>>>(x, y, out, N)
    )?;
    stream.synchronize()?; // Wait for kernel to finish
    let mut result: f32 = 0.0; // Prepare location for result
    cuMemcpyDtoH_v2(&mut result as *mut _ as *mut c_void, // copy result
                    out.as_raw() as u64 + 37, size_of::<f32>());
    println!("Sum is {}", result);
    Ok(())
}}
```

Listing 4.4: Example of using CUDA in Rust.

tions over the C API. The general idea of this package is equivalent of the one used with OpenCL, but slightly less mature.

The structure of the program is the same as the OpenCL example. It starts with the initialisation on Lines 8 to 30. Prepares the GPU memory on Lines 20 to 24. Then it launches the kernel on Line 26, and waits for the GPU to finish on Line 29. Finally Line 31 copies one of the results from the GPU memory to the host memory, and prints this result on Line 33.

Contrary to the OpenCL example, this CUDA program needs to be compiled into intermediate bytecode. This compilation is performed by `nvcc`, of which the result is used in the example. After the intermediate bytecode is generated, the program can be run using the following command: `cargo run`. The code can be cross compiled for most platforms, and for each the Rust compiler can provide a single binary binary file. However, the code will not run on platforms lacking an NVIDIA GPU, making the distribution of this software more difficult.

We choose to employ CUDA in our prototype, because of the higher level of support provided by NVIDIA. This makes the development easier, and faster.

Chapter 5

Data structures

Chapter 3 introduced IPOG, and mentioned various data structures. The most important ones are the MCA, and the CM. We will discuss the implementation of these data structures in our prototypes in this chapter. During this discussion we will explore various alternatives of each data structure. We can increase the overall performance of our prototypes by selecting the best data structures, which will show us the full potential of the acceleration techniques discussed in Chapters 6, 7 and 8. In Section 5.1 we introduce the implementation of the CM. Section 5.2 will introduce the implementation of the MCA. However, we first introduce the "index" function.

The "index" function is used when querying information in the proposed data structures. This function uses the lookup tables in Figure 5.1 to retrieve the index of the given argument. For example, if we have the PC $\langle P_{\text{Browser}}, P_{\text{CPU}} \rangle$, then we define the retrieval of the index as follows:

$$\text{index}(\langle P_{\text{Browser}}, P_{\text{CPU}} \rangle) = 1$$

Value	Index
P_{CPU}	0
P_{Browser}	1
P_{OS}	2

(a) Lookup table for the parameters.

Value	Index
$\langle P_{\text{OS}}, P_{\text{CPU}} \rangle$	0
$\langle P_{\text{Browser}}, P_{\text{CPU}} \rangle$	1

(b) Lookup table for the PCs.

Value	Index
$\langle V_{\text{CPU}}, \text{AMD} \rangle$	0
$\langle V_{\text{CPU}}, \text{Intel} \rangle$	1
$\langle V_{\text{Browser}}, \text{Firefox} \rangle$	0
$\langle V_{\text{Browser}}, \text{Chrome} \rangle$	1
$\langle V_{\text{OS}}, \text{Linux} \rangle$	0
$\langle V_{\text{OS}}, \text{Windows} \rangle$	1
$\langle V_{\text{OS}}, \text{Mac} \rangle$	2

(c) Lookup table for the values of parameters.

Figure 5.1: The values of each type have been given a unique index, which are shown in this lookup table. The data structures shown in Figures 5.3 and 5.4 use these indices to query coverage information of an interaction.

		V_{CPU} AMD	Intel
V_{OS}	Linux	C_1	C_2
	Windows	C_3	C_4
	Mac	C_5	C_6
V_{Browser}	Firefox	C_7	C_8
	Chrome	C_9	C_{10}

Figure 5.2: This CM is used to visualise the link with the data structures in Figures 5.3 and 5.4. Each cell can be found in these data structures.

Similarly, if we have the value $\langle V_{\text{OS}}, \text{Linux} \rangle$, then we retrieve the index like so:

$$\text{index}(\langle V_{\text{OS}}, \text{Linux} \rangle) = 0$$

5.1 Coverage-map

The Coverage-map (CM) will need to keep track of the interactions covered by the current MCA. During IPOG we will access this information repeatedly, which means that choosing the correct data structure is of major importance. In Chapter 3 we use a table to represent the CM. Examples of such a table can be found in Figures 3.2.b and 3.3.b. In Figure 5.2 we copied this table and numbered every cell. In the proposed implementations, the state of these cells will be represented by a value in memory.

In this section we will compare three implementations of this data structure. We will describe the first implementation in Section 5.1.1. This data structure is a tree, which is easy to use but requires several random memory lookups potentially slowing it down. In Section 5.1.2 we will describe an array based data structure. It is harder to implement, but reduces the number of random memory lookups. Section 5.1.3 will explain how these two implementations can be combined. This implementation has less random memory lookups than the tree like implementation and simplifies the computation required to access values.

5.1.1 Tree structure

Figure 5.3 shows a visual representation of the tree like data structure. The bottom row contains the booleans or bits representing the coverage state of the interactions. Rows above this contain pointers, these are visualised by the arrows exiting these cells. Each touching set of touching cells represent a dynamically created array. We will use the index of the PC to get a pointer in

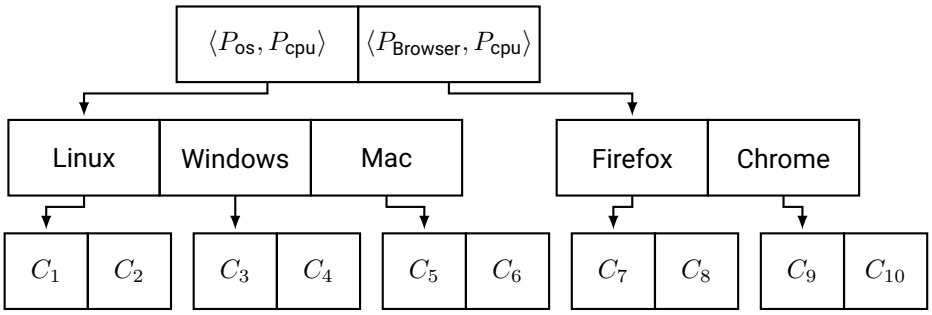


Figure 5.3: This figure visualises the proposed tree like implementation. Cells with outgoing arrows contain pointers to the node to which the arrows points. The bottom row shows cells with the coverage state of interactions. Coverage state of an interaction can be accessed by traversing the tree. Not to scale.

the top row. This pointer references one of the arrays on the second row. We use the index of the value to get the pointer to the next array. Finally, we use the index of the value of V_{CPU} to get to the memory address of the coverage state of the interaction.

For example, say we want to get the state of the following interaction:

$$(\langle V_{\text{Browser}}, \text{Firefox} \rangle, \langle V_{\text{CPU}}, \text{Intel} \rangle)$$

Then we first get the index of the parameter interaction:

$$\text{index}(\langle P_{\text{Browser}}, P_{\text{CPU}} \rangle) = 1$$

We use this index to get the pointer to the next array from the array at the top of Figure 5.3. Following the pointer we get to the array containing "Firefox" and "Chrome". We use the index of the value for the first parameter of the PC to get the next pointer:

$$\text{index}(\langle V_{\text{Browser}}, \text{Firefox} \rangle) = 0$$

Following the pointer obtained using the index, we arrive at the array containing C_7 and C_8. The value of the next parameter provides us with the following index:

$$\text{index}(\langle V_{\text{CPU}}, \text{Intel} \rangle) = 1$$

We access the cell at this index in the array, and find C_8. Figure 5.2 confirms that this is indeed the correct cell.

The process of looking up values is rather easy, but it requires several random memory lookups. Building the structure is also easy, but creating and clearing hundreds of arrays might be slow. After each iteration of IPOG, the program will need to walk through the tree to clear all boolean arrays to prepare for the next iteration.

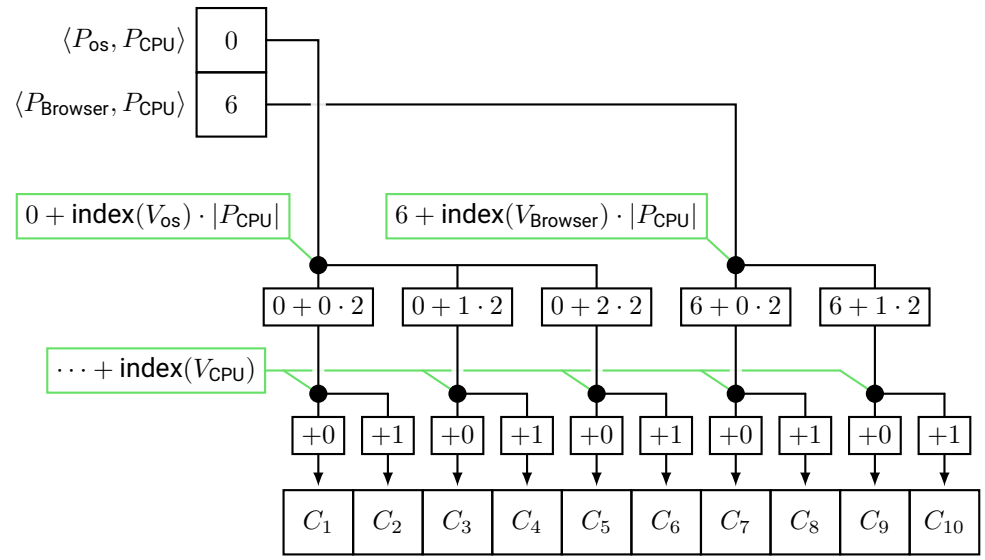


Figure 5.4: This figure visualises the proposed array implementation. The bottom row shows cells with the coverage state of interactions. To retrieve a cell in this array we calculate an index. To do so we start at the top, and work our way down following the correct line. Not to scale.

5.1.2 Array structure

Figure 5.4 shows a visual representation of the array data structure. At the bottom a continuous array is shown. To retrieve a cell in this array we need an index, which we calculate using various properties of the interaction. We start the calculation of the index by using the array at the top of the figure. Depending on the PC of the interaction you start with either 0 or 6. Then we add the index of the first value multiplied by the product of the length of the subsequent parameters. We do this until we used all values of the interaction. Finally, we use the resulting number as an index to the large continuous array.

For example, say we want to get the state of the following interaction:

$$(\langle V_{\text{Browser}}, \text{Firefox} \rangle, \langle V_{\text{CPU}}, \text{Intel} \rangle)$$

Then we first get the index of the parameter interaction:

$$\text{index}(\langle P_{\text{Browser}}, P_{\text{CPU}} \rangle) = 1$$

We use this index to get the starting value from the array at the top of Figure 5.4, which is 6. Following the line from this cell we arrive at the first addition statement, which we resolve as follows:

$$6 + \text{index}(\langle V_{\text{Browser}}, \text{Firefox} \rangle) \cdot |P_{\text{CPU}}| = 6 + 0 \cdot 2 = 6$$

We follow the line marked with "$6 + 0 \cdot 2$", and arrive at the next step. At the next step we add the index of the last value of the interaction:

$$6 + \text{index}(\langle V_{\text{CPU}}, \text{Intel} \rangle) = 6 + 1 = 7$$

We access the cell at this index in the array, and find C_8. Figure 5.2 confirms that this is indeed the correct cell.

The process of looking up values is more difficult than in the tree like implementation. However, memory lookups tend to be slower than computations. Creating the array is easy, as it is simply a big block of cleared memory. Clearing the CM for the next iteration of IPOG is easy, as we can simply wipe the entire memory block.

5.1.3 Combined implementation

The two implementations can also be combined to form an implementation with the advantages of both worlds. We take the top level of the tree like structure and have the pointers point to a continuous array each. This gives us a tree with two levels. We slightly simplify the index calculation compared to the single continuous array, and we simplify the clearing of the CM compared to the tree implementation. However, the ease of implementation is slightly reduced as we have to combine multiple techniques.

5.1.4 Common implementation details

In pseudocode of IPOG, found in Listings 3.1 and 3.2, the algorithm checks whether the entire CM is covered. The easiest way to know whether the CM is covered is by checking all values in the CM. However, this is rather slow, so instead we propose to keep track of the total number of uncovered interactions. If this number is equal to zero, then the entire CM is covered. Every time an interaction is covered we will need to update this value accordingly.

The CM needs to keep track whether an interaction is covered. This can be stored in a boolean. However, booleans are inefficient when it comes to memory usage, as they take up one byte each. A bit set is a data structure, where each bit is used to represent one distinct value. This means that the memory usage can be up to eight times less when using a bit set. However, the tree like implementation will not always be able to use this advantage. This can be seen in the visualisation found in Figure 5.3. All the arrays have only two elements, so instead of two bytes for the booleans, the bit set will use at least one byte. This is only two times less memory usage, so the bit set might not be suitable for the tree like implementation. In the following section there are two implementations using this bit set.

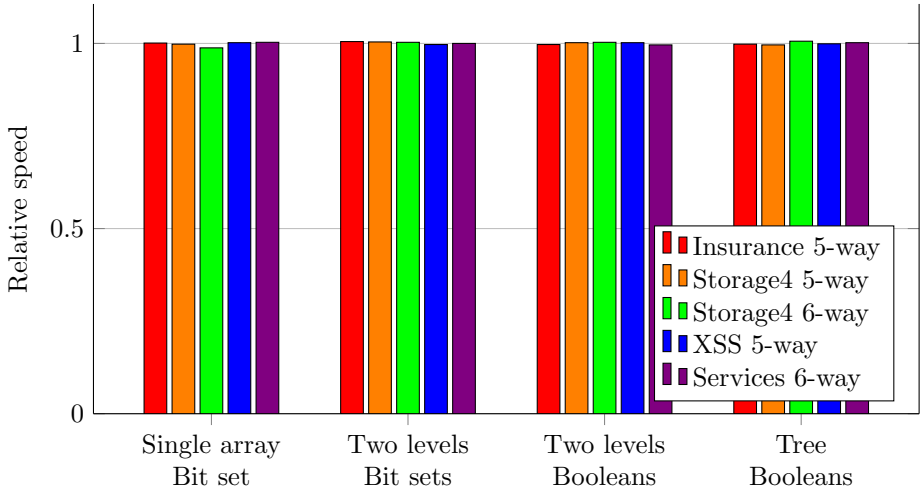


Figure 5.5: This chart provides a relative comparison of the various tested implementations. We selected five SUTs and ran the program multiple times using each implementation. Each bar represents the speedup relative to the average runtime of all implementations for the benchmark. A relative speed of 1 is equal to the average, lower is faster.

Benchmark	Single array Bit set	Two levels Bit set	Two levels Booleans	Tree Booleans
Insurance 5-way	1.00	1.00	1.00	1.00
Storage4 5-way	1.00	1.00	1.00	1.00
Storage4 6-way	0.99	1.00	1.00	1.01
XSS 5-way	1.00	1.00	1.00	1.00
Services 6-way	1.00	1.00	1.00	1.00

Figure 5.6: This table provides the relative speeds of various implementations solving various benchmarks. We selected five SUTs and ran the program multiple times using each implementation. Each value represents the speedup relative to the average runtime of all implementations for the benchmark. The relative speed of 1 is equal to the average, lower is faster.

5.1.5 Comparing performance

In Figure 5.6, we show the performance of each implementation relative to the average runtime of all implementations for the benchmark. Figure 5.5 visualises this information. In the figures, a score of 1 is equal to the average, and a lower score is faster. The tree implementation is described in Section 5.1.1. Section 5.1.3 described the two leveled implementations. In Section 5.1.2, we described the single array implementation. Each implementation will either

use arrays of booleans, or use the bit sets discussed in Section 5.1.4.

From the figures we can conclude that switching implementations does not result in changes in performance. Using bit sets instead of arrays of booleans also does not have measurable effects. Instead of looking at the performance of the different methods we will have to look at different metrics. We considered the memory usage, and ease of implementation. Memory usage is important, as a reduced usage allows for the generation of MCAs of bigger SUTs. The bit set implementations are more memory efficient, and the tree implementation can not fully take advantage of this property due to the limited size of the many arrays. So, the two level implementation and the single array implementation are the best at memory efficiency. Between the two, the single array implementation is the easiest to implement. That is why we use the single continuous bit set in our prototypes.

5.1.6 The list of Parameter Combinations

In all above mentioned implementations there is a need to assign an index to an interaction. Figure 5.7 shows an assignment of indices to PCs. It is a lookup table for PCs with 4 parameters, where each column represents a PC. However, each column only selects 3 parameters. The PCs considered always contain the to be added parameter. We choose to add this parameter whenever it is needed, because this allows us to use the lookup table for the next iteration of IPOG without modification. As the table in the figure shows, the PCs used to add P_4 is a subset of the combinations used to add P_5. This is possible since we did not include the last parameter in the lookup table.

It is also possible to calculate the index from the PC itself. To do this we define the following index assignment:

$$\text{index}(P_n) = n$$

We use this definition to calculate the index of a PC as follows:

$$\begin{aligned}\text{index}(\langle P_0, P_2, P_4, P_5\rangle) &= \binom{\text{index}(P_0)}{1} + \binom{\text{index}(P_2)}{2} + \binom{\text{index}(P_4)}{3} \\ &= \binom{0}{1} + \binom{2}{2} + \binom{4}{3} = 0 + 1 + 4 = 5\end{aligned}$$

If we use the lookup table (Figure 5.7), then we see that this is indeed the index we assigned to the combination.

Getting from an index to a PC is more difficult. We first define the following:

$$i,\, j,\, k \in \mathbb{N}_0 \quad i < j < k$$

	Index									
Parameter	0	1	2	3	4	5	6	7	8	9
P_0	×	×	×		×	×		×		
P_1	×	×		×	×		×		×	
P_2	×		×	×		×	×			×
P_3		×	×	×				×	×	×
P_4					×	×	×	×	×	×
P_5										
				To add P_4						**To add** P_5

Figure 5.7: Each column of this lookup table represents a Parameter Combination (PC). This table is for PCs with 4 parameters. During each iteration of IPOG, the parameter to be added is included as the last parameter of the PC in this table. Each successive iteration will have additional PCs, which has been visualised by the dashed boxes. The PCs are listed in colexicographical order. This table is for a SUT with 6 parameters and a 4-way MCA.

With this definition we get the PC assigned to index 5:

$$\begin{aligned}
\text{index}^{-1}(5) &= \langle P_i, P_j, P_k, P_5 \rangle \\
5 &= \binom{i}{1} + \binom{j}{2} + \binom{k}{3} \\
&= \frac{i!}{(i-1)!\,1!} + \frac{j!}{(j-2)!\,2!} + \frac{k!}{(k-3)!\,3!} \\
&= \frac{i}{1} + \frac{j\,(j-1)}{2} + \frac{k\,(k-1)\,(k-2)}{6}
\end{aligned}$$

The above formula is solved by a SAT solver or brute force. Both methods are costly, so we avoid this when implementing our prototype.

There are two ways to avoid the need to calculate the inverse of the index. We already discussed the lookup table, which can provide the PC within a single memory lookup. We can also reuse the algorithm to generate the lookup table. During the run of IPOG we will repeatedly iterate through all PCs. Calculating the next PC is inexpensive, and increasing the index is also simple. Further research could investigate to find out which solution is faster. The lookup table is far easier to implement compared to the calculation, so for now we go with the table.

As mentioned above, the list of PCs of an iteration of IPOG are used again in the next. We order the PCs in such a way that we only need to generate the entire table once. If the SUT contains n parameters, we generate the table for the nth parameter. For each iteration of IPOG we use an increasing portion of

this table. For example, if we have 6 parameters and generate a 4-way MCA then we generate the table as shown in Figure 5.7. During the first iteration of IPOG we add the 5th parameter, which is P_4 in our example. There are $\binom{4}{3} = 4$ PCs of the 4 previous parameters, so we use the first 4 PCs of our lookup table. The figure also shows this partition using a dashed box, marked with the text "**To add** P_4". In the next iteration of IPOG, which is also the last, we use the first $\binom{5}{3} = 10$ PCs. This partition is shown as a dashed box in the figure, marked with the text "**To add** P_5".

The time saved using this generation method is not by any means significant. However, it is simple to implement, so we use this in our implementation. Another advantage of this method is that we do not need to create a GPU version of the lookup table generation. We can simply use the CPU based generation and copy the result to the GPU memory. The GPU code can then use this table for the entire run of IPOG.

5.2 Mixed-level Covering Array

We also need a data structure to represent an Mixed-level Covering Array (MCA). As discussed in Section 2.2, an MCA is a list of tests. Each test has values assigned to the parameters of the SUT.

We compare possible implementations, as we did for the CM in the previous section. One implementation has an array, where each cell has a separate array representing a single test. We will refer to this implementation as a two-dimensional array, as access requires two separate indices to access the assigned values. Both dimensions are of unknown sizes and change between/during runs of the algorithm, so we use `Vec`s for both. A `Vec` is the list type of Rust that allows for dynamic resizing. The `array` is the list type of Rust that does not allow for resizing.

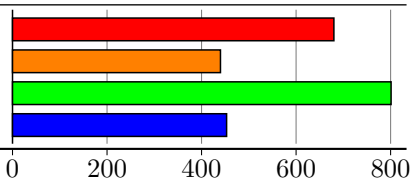
The other implementation has one continuous array where every cell represents a value in the MCA. Indexing this array requires the transformation of the test number and parameter to an index as follows:

$$\text{index}(\text{test}, P_n) = \text{index}(\text{test}) \cdot |P| + \text{index}(P_n)$$

We will refer to this implementation as a one-dimensional array. The size of this array changes during a run of the algorithm, so we use a `Vec`.

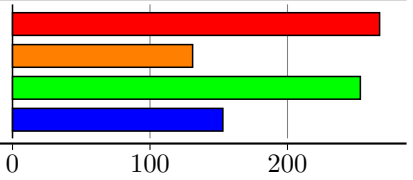
If we know the size of the SUT in advance, then we can compile the arrays using this size. For the two-dimensional array, we use an `array` instead of a `Vec` for the second dimension. The first dimension must allow for additional rows (added during VE), so this stays a `Vec`. The one dimensional array must also allow for additional values, so this stays a `Vec`. However, when iterating over the MCA, we can do so per row without additional boundary checks. This allows for a similar performance as an `array`. The versions with a known SUT

MCA type	**Time** (ns)
One-dimensional dynamic	680
One-dimensional static	440
Two-dimensional dynamic	801
Two-dimensional static	453



(a) The benchmark providing these results reads all the values in the prototype data structure, testing the efficiency of sequential traversal of the MCA.

MCA type	**Time** (ns)
One-dimensional dynamic	267
One-dimensional static	131
Two-dimensional dynamic	253
Two-dimensional static	153



(b) The benchmark providing these results reads all values of various rows all over the MCA.

Figure 5.8: These tables show benchmark results of the various proposed implementations of the MCA.

size will be referred to as “static”. If the size is unknown, then we will refer to it as a “dynamic” implementation.

We have static and dynamic versions of the two- and one-dimensional arrays. In total that gives us four implementations to compare.

We compare the implementations using two benchmarks. The results of these benchmarks can be seen in Figure 5.8. First, we test the sequential read of all values in the MCA, copying behavior seen during the HE. Figure 5.8.a shows the results of this benchmark. Next, we test the random read of rows in the MCA, copying behavior seen during the VE. The results of this benchmark can be seen in Figure 5.8.b.

The results of the benchmarks allow us to draw the conclusion that the static versions are superior. There is, however, a drawback to this prototype. As mentioned in Section 4.2, compiling the prototypes for multiple variables takes time. The prototypes would need to be compiled for all the possible SUT sizes. The SUTs we use to test have sizes ranging from 5 to 35 parameters (see Figure A.1), so the compile time would get about 31 times longer. Currently, the compile time is at least 30 seconds. Compiling with the SUT sizes precompiled would take about 15 minutes. While developing, this would be inconvenient.

There are two solutions to reduce this compilation time. One solution is that we only compile one size during development and only add support for the other sizes in the final version. We would keep the current compile time,

but we lose the ability to test the prototypes with various SUTs. Another solution is to compile on demand, so only to compile the prototype for the specific size once the program has been called. Subsequent calls with SUTs of the same size can reuse the resulting compiled code. It can take hours to generate the MCA for a larger SUT at a high strength, so the additional compile time could pay off if the prototype generates the MCA faster.

Future research could provide us with an answer as to which solution works best. For now, we use one-dimensional dynamically sized MCA, as the implementation is easier. For both the CPU based prototypes and the GPU based prototype.

Chapter 6

Acceleration using bitwise operations

In this chapter we will answer the following question: *How do we accelerate the IPOG algorithm using bitwise operations?* We will introduce alternate representations of the data used in IPOG. These representations allow for the use of bitwise operations. These bitwise operations replace existing statements in the algorithm, reducing the total number of operations during a run of the algorithm. The reduction of the total number of operations results in the acceleration of the IPOG algorithm.

This chapter will introduce two new data structures. The first of the data structures is the don't-care locations, which is a partial representation of a row in the MCA. The other is the parameter locations, which represents a PC.

We will first provide further background on these data structures in this chapter. Then, in Section 6.1, we will discuss how to use these data structures during the HE. After that, in Section 6.2, will discuss how we use them during the VE. Finally, we evaluate the application of this technique on both the HE and VE in Section 6.3.

The don't-cares are an important part of the inner workings of the IPOG algorithm. Profiling the prototypes show that the code handling these values takes considerable time. We propose the use of an additional data structure to supplement the MCA discussed in Section 5.2. This data structure is a partial representation of a row in the MCA, and will be called the don't-care locations. It is a bit set where each bit represents the presence of a don't-care at the equivalent index in the row of the MCA. Figure 6.1 shows an example of a few rows with their respective bit sets. Each row on the left has various values, including don't-cares. A don't-care is represented by a "*", in the row. The locations values are shown as bit sets, where each bit is set to "1" if there is a don't-care at that index of the row.

The parameter locations of a PC is a bit set, where each bit represents the presence of a parameter in the PC. There is an exception though, as the last parameter in the PC is ignored. The last parameter is the one to be added

Rows of an MCA							Don't-care locations
1	1	1	0	1	0		000000_2
2	*	0	*	0	1	$\Longrightarrow$	010100_2
3	0	*	*	1	*		001101_2
*	*	*	*	*	*		111111_2

Figure 6.1: The left table shows various rows, part of an MCA. The cells with a "*" are don't-cares. The right table with Don't-care locations contains bit sets representing the locations of don't-cares in the rows shown in the table on the left.

PC		Parameter locations
$\langle P_0, P_1, P_5 \rangle$		110000_2
$\langle P_1, P_4, P_5 \rangle$	$\Longrightarrow$	010010_2
$\langle P_2, P_3, P_5 \rangle$		001100_2
$\langle P_2, P_4, P_5 \rangle$		001010_2

Figure 6.2: This table provides an example of the parameter locations of a PC for a 3-way generation. The index of each parameter in the PC is used to set the corresponding bit in the parameters locations variable. However, the last parameter is always the parameter added in the latest iteration of the IPOG algorithm, so we leave this parameter out of consideration.

during the current iteration of the IPOG algorithm. This makes the presence of the parameter a given, which is why we can safely ignore it in the parameter locations. Figure 6.2 provides a selection of PCs and their corresponding parameter locations.

In Sections 6.1 and 6.2 we will explain how the proposed data structure can be used to speed up each of the extensions. These extra data structures have been used to accelerate both the single- and multithreaded prototypes. Further research will be needed to find if this technique is also effective in the GPGPU prototype.

6.1 Bitwise in Horizontal Extension

In this section we will discuss how we can use this data structure to accelerate the score calculation during HE. During HE, the MCA is extended with an additional parameter. Each existing row will get a value for this parameter. We consider the best value for each row to be the value that covers the most yet uncovered interactions. Counting the uncovered interactions for each value requires a lookup in the CM for each PC. There are a lot of these PCs, resulting in a lot of lookups in the MCA. Using the extra data structure we attempt to

	Examples			
	A	**B**	**C**	**D**
Don't-care locations	001101_2	001101_2	001101_2	001101_2
Parameter locations	110000_2	010010_2	001100_2	001010_2
Bitwise and	000000_2	000000_2	001100_2	001000_2

Figure 6.3: Some examples of don't-care locations and parameter locations have been provided in this table.

make these lookups more efficient. We will first introduce the original algorithm, then we will explain what modifications we made.

The improvement works with any of the proposed CM implementations mentioned in Section 5.1. We use the tree-like CM in our explanation, as its simpler lookup code makes the pseudocode easier to read. The original version of the lookup algorithm for this tree-like CM has been listed in Listing 6.1.a. It consists of two functions, named `get_score_checked` and `lookup_checked`. The function `get_score_checked` iterates over all PCs (Line 3), queries the interaction (Line 4), and counts every uncovered interaction (Line 5). It uses the function `lookup_checked` to query the interaction in the CM.

The `lookup_checked` function traverses the tree-like CM. It starts on Line 12, using the PC index to get to the next node. The loop starting on Line 13 iterates over every parameter index in the PC. This index is then used to get the chosen value for this parameter on Line 14. Line 15 checks whether the value is a don't-care, and bails if it is. We bail because HE only changes the value for the newly added parameter. It can not cover interactions using don't-cares for other parameters. If it does not bail, then it goes to the next node in the tree using the value. The last node is returned on Line 18.

The new version differs with only a few lines. These differences have been highlighted. Line 4 of Listing 6.1.a is equivalent to Line 5 in Listing 6.1.b. Line 15 of Listing 6.1.a is replaced by Line 4 in Listing 6.1.b. In essence, the check for a don't-care value is moved from the lookup method to outside. This means that the lookup method is not called if the PC has a don't-care in the row. Line 4 of Listing 6.1.b checks for the presence of a don't-care in the row overlapping with the parameters in the PC.

If a don't-care in the row overlaps with a parameter in the PC, then the don't-care locations and parameter locations share a "1" at the same index. We can perform a bitwise "and" to find all overlapping values:

$$\text{don't-care locations} \mathbin{\&} \text{parameter locations} = \text{overlap}$$

If the "overlap" has any bit set, then there is an overlap in don't-cares and parameters. Otherwise, no overlap exists and the PC can be used to count the

```
1  fn get_score_checked(row, cm, new_value) -> int {
2      let score = 0;
3      for pc in PARAMETER_COMBINATIONS_CACHE {
4          if let Some(sub_cm) = lookup_checked(row, cm, pc) {
5              if !sub_cm[new_value] { score += 1; }
6          }
7      }
8      return score;
9  }
10
11 fn lookup_checked(row, cm, pc) -> Option<Vec<bool>> {
12     let mut cm = cm[pc.index];
13     for parameter_index in pc.parameters {
14         let value = row[parameter_index];
15         if value == DONT_CARE { return None; }
16         cm = cm[value];
17     }
18     return cm;
19 }
```

(a) The original algorithm.

```
1  fn get_score_prechecked(row, cm, new_value) -> int {
2      let mut score = 0;
3      for pc in PARAMETER_COMBINATIONS_CACHE {
4          if row.dont_care_locations & pc.parameter_locations == 0 {
5              let sub_cm = lookup_unchecked(row, cm, pc);
6              if !sub_cm[new_value] { score += 1; }
7          }
8      }
9      return score;
10 }
11
12 fn lookup_unchecked(row, cm, pc) -> Vec<bool> {
13     let mut cm = cm[pc.index];
14     for parameter_index in pc.parameters {
15         let value = row[parameter_index];
16         cm = cm[value];
17     }
18     return cm;
19 }
```

(b) The new algorithm.

Listing 6.1: Both of the pseudocode listings above provide the caller with the number of newly covered interactions for a given `new_value`. The original version checks for the existence of don't-cares in the lookup sub function, on Line 15 of Listing 6.1.a. The new version checks on Line 4 of Listing 6.1.b, before calling the sub function. Differences have been highlighted.

```
1  fn get_score_split(row, cm, new_value) -> int {
2      if has_dont_cares(row.dont_care_locations) {
3          return get_score_prechecked(row, cm, new_value);
4      } else {
5          let mut score = 0;
6          for pc in PARAMETER_COMBINATIONS_CACHE {
7              let sub_cm = lookup_unchecked(row, cm, pc);
8              if !sub_cm[new_value] { score += 1; }
9          }
10         return score;
11     }
12 }
```

Listing 6.2: The pseudocode listed here is an alternate version the algorithm listed in Listing 6.1.a. It extends `get_score_prechecked` with a special case. If the given row has no don't-cares then there is no need for any additional don't-care checks, so these are skipped. The effects of this difference on the performance have been illustrated by Figure 6.4.b.

number of interactions covered by the new value.

Figure 6.3 shows the don't-care locations of the third row in Figure 6.2 and the parameter locations in Figure 6.2. The last rows shows the result of the bitwise "and" between the two locations. If the result is equal to 000000_2 then the PC can be used, and the lookup function is called to retrieve the coverage information of the interactions.

If there are no don't-cares in the row, then we need not check for overlapping don't-cares with the parameters in the PC. Using this insight results in an alternate version of the score calculation algorithm. The pseudocode for this version has been listed in Listing 6.2. Line 2 of this listing checks for the existence of don't-cares in the row. If any exist then Line 3 calls the `get_score_prechecked` function of Listing 6.1.b. The absence of don't-cares allows us to remove the check for them. Lines 5 to 10 of Listing 6.2 are equivalent to the `get_score_prechecked` function of Listing 6.1.b, but without the check for the existence of don't-cares.

Checking for the existence of don't-cares in the row is equivalent to the check with the parameter locations of the PCs. However, in this case all parameters are checked for overlapping values. As an example we use our example row from Figure 6.3 with don't-care locations 001101_2. We take the parameter locations for all parameters, which is 111110_2. Then we bitwise "and" the two values:

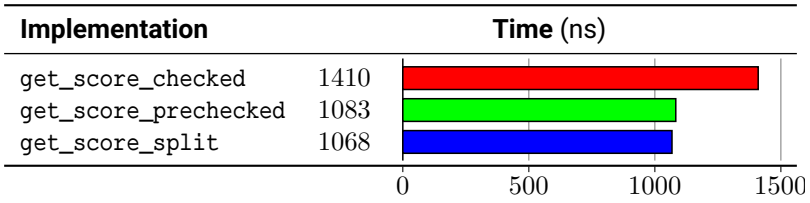
$$001101_2 \,\&\, 111110_2 = 001100_2$$

This results in value other than 000000_2, so the row has don't-cares. If we take a row without don't-cares, then the don't-care locations is 000001_2. The

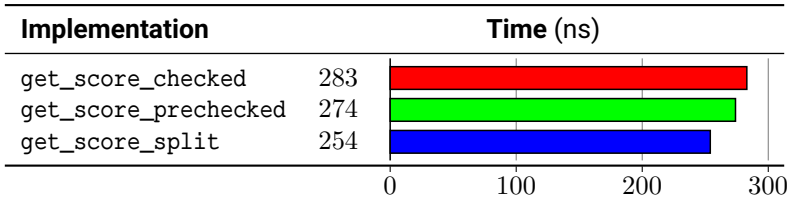
bitwise “and” with this value is:

$$000001_2 \,\&\, 111110_2 = 000000_2$$

The result is equal to 000000_2, so we know that we can skip all checks for don’t-cares.



(a) Benchmark using 42 rows of various lengths, using parameters of various cardinalities, and filled with a variable number of don’t-cares.



(b) Benchmark using 6 rows of various lengths, using parameters of various cardinalities, and filled with no don’t-cares.

Figure 6.4: These benchmarks show the performance of various implementations of the scoring algorithm. The bitwise implementations use the extra data structure, and the switched version uses an alternate algorithm if no don’t-cares are present in the row.

Figure 6.4 shows the results of the benchmarks testing the efficiency of the three scoring functions listed in Listings 6.1 and 6.2. The first set of benchmarks shows the total time spend on calculating the score of 42 rows of various lengths, using parameters of various cardinalities, and filled with a variable number of don’t-cares. From the results we can conclude that both of the new versions outperform the original scoring function. The difference between the new versions is not big enough to draw any conclusions, which is why the second set of benchmarks in Figure 6.4.b is provided. These benchmarks are the subsets of the ones listed in Figure 6.4.a. Only the rows without don’t-cares have been tested in these benchmarks. This makes the difference in performance clearer. The difference is not big, but it can add up over the course of the generation. That is why we use the `get_score_split` function, listed in Listing 6.2, in our single- and multithreaded prototypes. We need more research to determine the viability of this improvement when applied to the GPGPU version. It is likely that the branch divergence of all versions of the scoring functions have a negative impact on the performance, so the difference in overall performance is likely to be insignificant.

6.2 Bitwise in Vertical Extension

In Section 6.1 we discussed how the new data structures can improve HE. In this section we will provide an explanation as to how VE can benefit from these same data structures.

In Section 3.3 we explained that VE tries to fit interactions into the MCA. These interactions are the ones that have not been covered during the HE. VE tries to insert those interactions in one of the existing rows. An interaction can only be inserted into a row with at least one don't-care, if there are no don't-cares then the interaction would already be covered by that row. In [15], the authors use this property to create a list of rows with don't-cares. During VE, the algorithm limits its search to this list for rows to fit the interactions in.

Combining this same property with the don't-care locations and parameter locations we can further improve the performance of the VE. There should be at least one don't-care overlapping with the interaction, so there should be an overlap between the don't-care locations of the row and the parameter locations of the interaction. We can give an early verdict by checking for this overlap, which should improve the performance of the VE. Specifically, the `interaction_fits` function used in the pseudocode of VE should perform better. The pseudocode in Listing 3.3 shows its usage on Line 4.

Listing 6.3 provides the pseudocode for the `interaction_fits` function for both the original version and the version using the don't-care locations and parameter locations. The original version, shown in Listing 6.3.a, iterates over each value in the interaction on Line 2. For each of these values the corresponding value in the row is retrieved (Line 3). The value retrieved from the row should be equal to the value from the interaction or it should be a don't-care. This is checked on Line 4. The function returns `false` if the check fails on Line 5. If the value of the row is compatible with the value from the interaction then the next value is checked. Finally, if all the values are compatible, then the function will return `true` on Line 8.

The new version adds a check for the existence of don't-cares, similar to the improvement listed in Section 6.1. However, unlike HE, VE will need to filter out the cases where the interaction does not overlap with a don't-care in the row. The pseudocode in Listing 6.3.b shows the new version of the code. Lines 2 to 4 are responsible for the different behaviour of the new version. The `if` statement checks for the existence of don't-cares, by performing the same operation shown in the pseudocode of the improved HE in Listing 6.1.b on Line 4.

Just like the improved version of HE, the new VE will skip cases early on instead of discovering incompatibility at a later stage. In Figures 6.5 and 6.6, we show the effect of this improvement by comparing the performance of the original version to the new version using bit sets. These figures show the results of multiple benchmarks, limited to those taking 60 seconds or more to complete when using the original version. Each row in the table of Figure 6.5

Figure 6.5: This table shows the speedup resulting from the usage of the locations bit sets during the VE. The benchmark was performed on all SUTs listed in Figure A.1 while ignoring the constraints, generating MCAs for the strengths 2 to 12. The tests for which the original version takes longer than 60 seconds are listed here. All times are in seconds. Figure 6.6 visualises the data in this table.

			Horizontal Extension			Vertical Extension			Total		
	Benchmark	t	Bit.	Orig.	Speedup	Bit.	Orig.	Speedup	Bit.	Orig.	Speedup
1	Storage4	6	57	56	0.97	4	7	1.96	61	63	1.03
2	Storage3	9	20	20	1.00	75	87	1.17	95	108	1.13
3	Insurance	6	51	50	0.98	7	67	9.19	59	117	2.00
4	Services	7	9	8	0.92	62	144	2.33	71	153	2.15
5	Processorcomm1	10	42	41	1.00	276	398	1.44	318	439	1.38
6	Storage3	10	32	32	1.00	484	554	1.15	515	586	1.14
7	Xss	6	7	7	0.96	58	615	10.62	65	622	9.59
8	Coveringcerts	7	643	613	0.95	15	15	1.00	658	628	0.95
9	Healthcare4	6	652	638	0.98	1	2	1.17	654	640	0.98
10	Storage5	6	602	588	0.98	59	128	2.18	661	717	1.09
11	Processorcomm2	7	1248	1180	0.95	16	17	1.11	1263	1197	0.95
12	Healthcare3	7	1276	1208	0.95	4	4	1.06	1280	1212	0.95
13	Processorcomm1	11	59	59	1.00	1077	1285	1.19	1136	1344	1.18
14	Storage4	7	1361	1284	0.94	193	547	2.84	1554	1830	1.18
15	Storage3	11	34	34	1.00	1844	2283	1.24	1879	2317	1.23
16	Services	8	27	31	1.14	1877	2931	1.56	1904	2963	1.56
17	Insurance	7	532	498	0.94	353	2892	8.18	886	3391	3.83
18	Processorcomm1	12	56	57	1.02	4756	5538	1.16	4812	5595	1.16
19	Storage3	12	23	24	1.02	6148	8084	1.31	6171	8108	1.31
20	Services	9	87	87	1.01	9467	14344	1.52	9554	14432	1.51
	Geometric mean				0.98			1.88			1.45

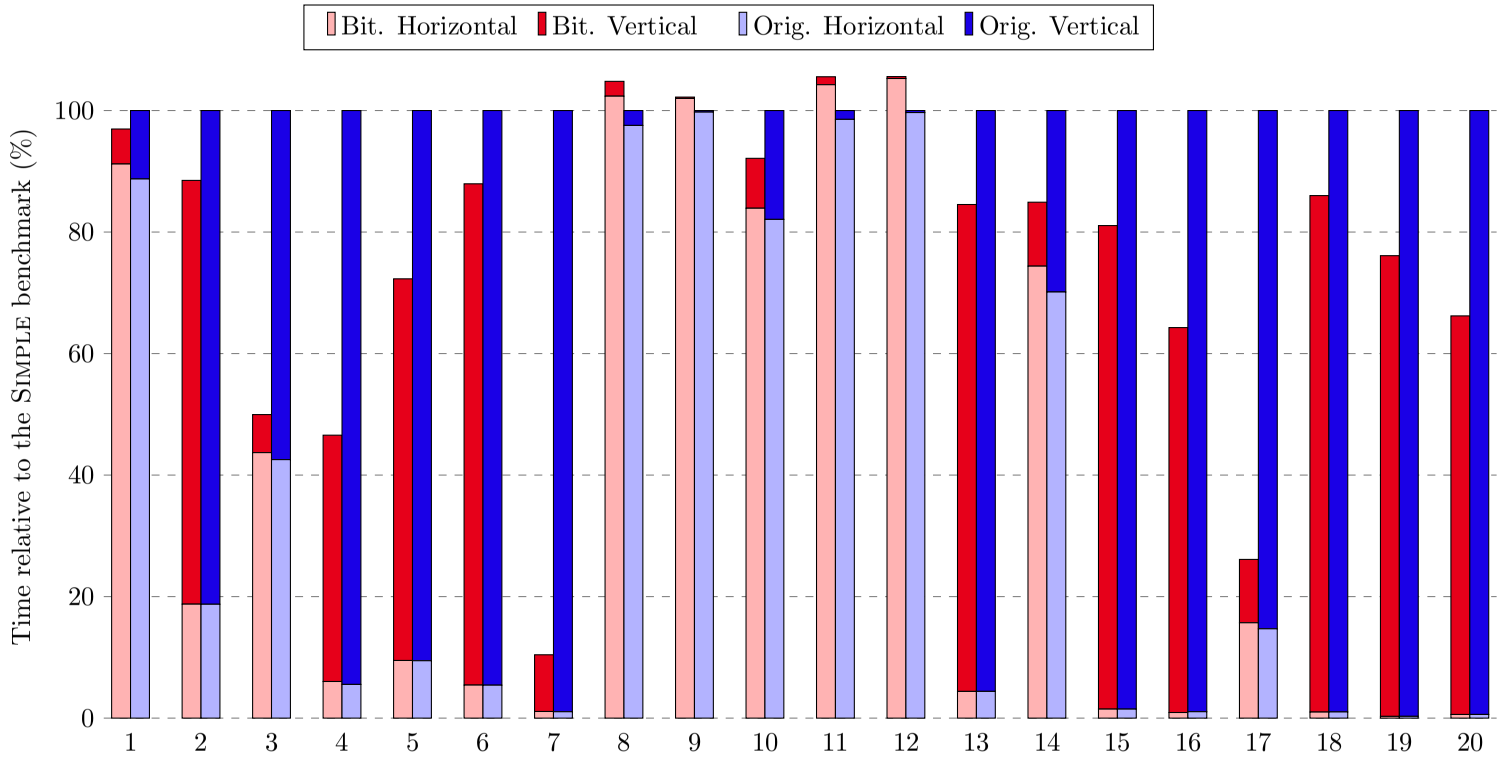


Figure 6.6: This chart visualises the data provided in Figure 6.5. Each pair of bars is numbered corresponding to the row in the table. A pair of bars shows the speeds relative to the original version. The red bar on the left shows the relative speed of the version using bit sets in the VE. The blue bar on the right represents the original version. Each bar is composed of two parts, where the top part represents the time taken by the VE. The bottom part of each bar represents the time taken by the HE.

```
1 fn interaction_fits(row, interaction) {
2     for parameter_index, interaction_value in interaction.values {
3         row_value = row[parameter_index];
4         if row_value != DONT_CARE && row_value != interaction_value {
5             return false;
6         }
7     }
8     return true;
9 }
```

(a) The original algorithm.

```
1  fn interaction_fits(row, interaction) {
2      if row.locations & interaction.parameter_locations == 0 {
3          return false;
4      }
5      for parameter_index, interaction_value in interaction.values {
6          row_value = row[parameter_index];
7          if row_value != DONT_CARE && row_value != interaction_value {
8              return false;
9          }
10     }
11     return true;
12 }
```

(b) The new algorithm.

Listing 6.3: Both of the pseudocode listings above check whether the given interaction can be fit into the given row. The original version checks for the existence of don't-cares in the lookup sub function, on Line 4 of Listing 6.3.a. The new version does the same on Line 7 of Listing 6.3.b, and also checks preemptively on Line 2 of Listing 6.3.b. The effects of this difference have been illustrated by Figures 6.5 and 6.6.

shows the time taken generating the given SUT at the given strength. A row shows the total time taken by each extension during the generation. It also shows the relative speedup of the new version compared to the original version. A relative speedup of 1.0 means an equivalent performance, anything less means a slow down, and anything more means a speedup compared to the original version.

These rows are visualised by the bars in Figure 6.6, where the row number matches the number under each pair of bars. A pair of bars shows the speeds relative to the original version. The red bar on the left shows the relative speed of the version using bit sets in the VE. The blue bar on the right represents the original version. Both of the bars in a pair are composed of two parts, where the top part represents the time taken by the VE. The bottom part of each bar represents the time taken by the HE.

The results show that the new version is as fast or faster than the original version during VE. However, the results also show that the new version is as slow or slower than the original version during HE. There are 4 benchmarks for which the slowdown of the HE is bigger than the speedup of VE. These can be found in rows 8, 9, 11, and 12. Note that the time taken by VE is relatively short compared to the time taken by HE. From this we can conclude that the new version is faster if the VE is more significant during the generation. The new version has a geometric mean speedup of 1.45 compared to the original version, so the new version performs better than the version.

The slowing down of HE can be explained by the extra work performed. The bit set of each row needs to be updated with each change to the row. Updating the data structure is fast, but not instant. This results in the slower runtimes of the HE. We solve this issue by also using the bit sets during HE.

Potentially we could further increase performance by reordering the order of execution during VE. Interactions sharing the same PC also share the same parameter locations. If the parameter locations fail the new check, then all the interactions sharing the same PC will not fit. However, the current implementation does not take this into account and performs the same check for all uncovered interactions. If we want to prevent the redundant checks, then we need to reorder the iteration order used during VE. Currently the order starts by iterating over the PCs, then iterates over the interactions, and finally iterates over the rows of the MCA. If we iterate over the rows before iterating over the interactions, then we can use the bit sets to determine if we need to iterate through the interactions in the first place. So the new order would be: PCs, rows of the MCA, and finally the interactions. Only one of the interactions sharing the same PC can fit in a row, so the resulting MCA should stay the same with this new order. Applying this new order requires a significant rewrite of the prototypes, and various improvements would need to be reimplemented and reevaluated. For example, the skips mentioned in [15] are likely to be impacted by this reordering. For this reason we will leave the implementation of this improvement to future research.

6.3 Evaluation

We compared the performance of our new bitwise operator using version of the single-threaded prototype to the original version. We generated 2-way to 12-way MCAs for all SUTs listed in Figure A.1 while ignoring the constraints. The runtime of both versions has been listed in Figure 6.7. All but one benchmark shows an improvement in performance for the new version. The geometric mean of the speedup is 1.52. This is higher than the speedup of 1.45 listed in Figures 6.5 and 6.6, which only measures the speedup of the bitwise VE. However, the difference is small. This means that the new bitwise HE increases the overall performance, but not by much.

			LibreCA		
	Benchmark	t	Bit.	Orig.	Speedup
1	Storage4	6	58	63	1.08
2	Storage3	9	92	108	1.17
3	Insurance	6	67	171	2.55
4	Services	7	61	177	2.91
5	Processorcomm1	10	272	362	1.33
6	Storage3	10	513	589	1.15
7	Healthcare4	6	603	637	1.06
8	Coveringcerts	7	624	638	1.02
9	Xss	6	90	737	8.20
10	Storage5	6	652	754	1.16
11	Processorcomm1	11	811	1016	1.25
12	Processorcomm2	7	1233	1196	0.97
13	Healthcare3	7	1185	1224	1.03
14	Storage4	7	1510	1840	1.22
15	Services	8	1533	2191	1.43
16	Storage3	11	1899	2421	1.27
17	Insurance	7	951	3818	4.02
18	Processorcomm1	12	4948	5872	1.19
19	Storage3	12	6620	8521	1.29
20	Services	9	7041	12890	1.83
	Geometric mean				1.52

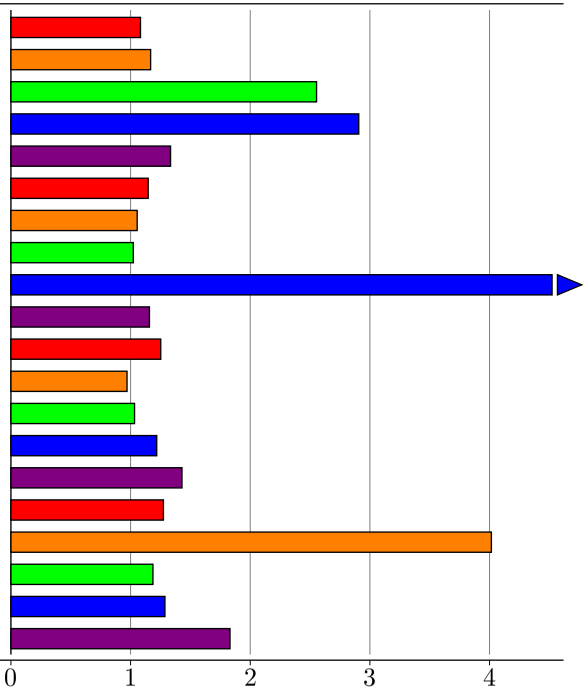


Figure 6.7: This table contains the results of the benchmarks measuring the performance of the bitwise and original versions of the single-threaded prototype. The benchmark was performed on all SUTs listed in Figure A.1 while ignoring the constraints, generating MCAs for the strengths 2 to 12. The “Speedup” column represents the relative speedup of the bitwise version compared to the original version. A speedup of 1 indicates equal speed, lower than 1 is slower, and higher than 1 is faster.

With a geometric mean of 1.52 we can conclude that the addition of the data structures proposed in this chapter accelerate IPOG.

The results allow us to answer the subquestion: *How do we accelerate the IPOG algorithm using bitwise operations?* We introduced the don't-care locations and the parameter locations as additional data structures. Using bitwise operators, we reduced the number of operations in the IPOG algorithm. This in turn reduced the runtime of our prototype, accelerating the IPOG algorithm.

Chapter 7

Acceleration using multithreading

This chapter will explain how multithreading can accelerate IPOG, answering the question: *How do we accelerate the IPOG algorithm using multithreading?* We start by explaining how the multithreaded prototype divides the tasks between the threads in Section 7.1. After that, we reduce the impact of imperfect task divisions by removing a direct dependency in Section 7.2. Next, we describe how the multithreading prototype dynamically alters the number of constraints handling threads in Section 7.3. Finally, we evaluate the multithreading prototype in Section 7.4.

7.1 Division of tasks when multithreading

We have three thread types in our multithreaded prototype during HE:

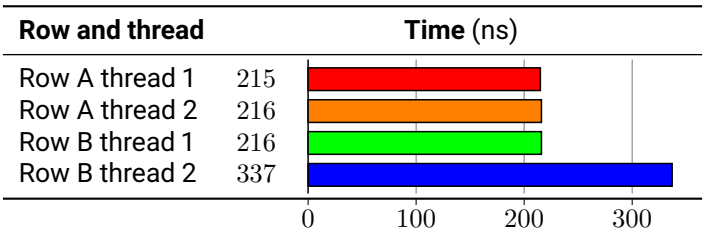
- **The main thread.** There is only one such thread in the program. It is responsible for the aggregation of results from the other threads. Only this thread can change the MCA and CM, which prevents concurrency related bugs. It also removes the need for any locking mechanism, which causes an overhead.
- **The worker threads.** These threads calculate parts of the score of each value during the HE.
- **The constraint thread.** This thread works on the constraints of the SUT.

In this section we will explore how we divide the score calculation of the values during the HE.

Dividing the tasks between threads should reduce the total runtime of the prototype. However, if the division of tasks is imperfect the prototype might not be faster. For example, one thread could take more time than others if the division of tasks is uneven. Our multithreaded prototype is only as fast as its

Row	Values													
A	0	0	0	0	0	0	*	0	*	0	0	0	0	*
B	*	*	*	*	*	*	0	0	0	0	0	0	0	*

(a) The rows used in the benchmarks.



(b) The resulting benchmarks.

Figure 7.1: This table shows the results of benchmarks testing the speed of score calculation for multiple rows. The PCs are split in two, so each thread will cover the same number of PCs. The split of PCs is the same over the two rows. However, the execution time of the benchmarks differs due to the number and position of don't-cares.

slowest thread. To prevent this we attempt to divide the work as evenly as possible. We divide the score calculation by assigning a subset of all PCs to each thread. Each thread will then calculate the score of each value for said PCs and pass these on to the main thread.

However, assigning PCs comes with a risk. A thread can take more time, depending on the number of don't-cares in the row. Figure 7.1 shows the result of benchmarks where the placement of don't-cares results in a thread taking more time. The division of PCs is the same between the rows, so the difference in runtime can only be explained by the don't-cares. This difference would not be a problem if the rows have randomised don't-cares, as the average runtime would be roughly equivalent over more rows. However, this is not the case when generating using the IPOG algorithm. Subsequent rows tend to have similar build. For example, the MCA is initially created by filling the first columns, which guarantees the absence of don't-cares in the first columns of the first rows. We generate PCs in order, which means that the list of PCs starts with PCs using only the first few parameters. The order of PCs also influences the don't-cares added during the VE, as the list of PCs is processed in order. This results in similar PCs to be relatively close to each other in the MCA. Both of these properties result in an unequal distribution of don't-cares in subsequent rows.

We attempt to avert the divergent execution times by rotating the subsets of PC between the threads. This rotation mixes up the weight of the tasks enough, reducing the chances of a thread getting more work than the others.

Additionally, the worker threads work ahead, as described in Section 7.2. This means that the threads have to have an equal amount of work on average, instead of an equal amount of work for every row.

7.2 Working ahead when multithreading

The multithreaded implementation spends time when communicating with other threads. Reducing this communication is key to further increase the performance, as communication itself does not further the computation of the MCA. We create fifo queues to save the results from the worker threads. The main thread takes results from all queues and aggregates the results.

However, changes to the MCA invalidates all results gathered before the change. This means that if thread A writes a new result to the queue, then this result would be invalid by the time the main thread reads it. The solution to this problem is to send intermediate results instead of the final results. In Listing 6.1.b we saw that the score calculation involves querying coverage information in the CM and counting the number of uncovered interactions. Instead of writing the number of uncovered interactions to the queue we write the pointers to the coverage information. The main thread retrieves the pointers from the queues and removes the outdated interactions. It can then count the interactions left, and select a new value for the next row. The pointers can then be used to update the CM, as all the values at the pointers need to be set.

There are multiple ways to remove outdated pointers. We test two types, and each type has two ways of removing the outdated pointers. The first type we discuss filters the pointers by keeping track of recently covered interactions. The pointers that are outdated have recently been changed to be covered in the CM. By keeping track of the history we can filter the outdated values. We call this method the history based filter.

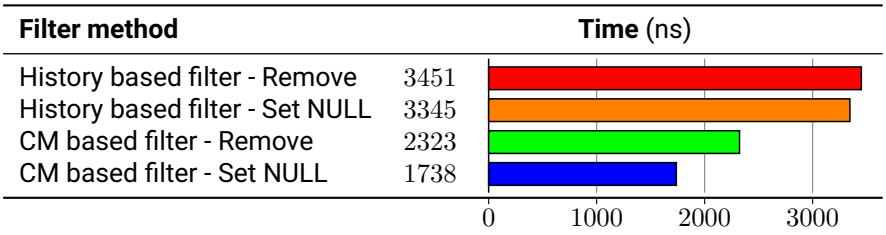


Figure 7.2: The results of the benchmarks in this table show the runtime of various methods for the filtering of outdated coverage information. The benchmarks include the filtering and setting the interactions as covered. They do not include the aggregation of pointers, as this would be the same for all methods.

The other type we discuss simply checks the current value by following the pointer. If it is covered, then the value can be removed. If the interaction is still uncovered, then the interaction is counted towards the score. We call this method the CM based filter.

The method of removal also influences the efficiency, so we test two methods. The first simply removes the pointer from the array. The other method sets the pointer to a reserved value, like NULL. If this value is later encountered it is skipped.

Figure 7.2 compares the 2 types both with the 2 removal methods. From the benchmark results we can conclude that the history based filter is slower than the CM based filter. We can also conclude that the removal of pointers is slower than setting the pointers to NULL. Skipping the NULL pointers takes extra time, but not as much as the removal of pointers from an list does. Removing from a list involves moving the next values in the list to their new indices, which is a relatively costly operation. In our multithreaded prototype we use the CM based filter.

7.3 Dynamic division of tasks

In Section 7.1 we discuss how the multithreaded prototype divides the work amongst the threads. The section also mentions the existence of a constraint solving thread. When generating MCAs for SUTs with constraints, this additional thread will use resources previously reserved for the worker threads. This will result in a unbalanced division of work amongst the threads, which in turn decreases the overall performance of the prototype. The number of the constraints present in the SUT will influence how much the performance decrease is.

Preliminary research shows that the number of worker threads should be adapted to the number of constraints. If the number of constraints is low, then we reduce the number of worker threads by 1. The CPUs with which we tested used hyper-threading, which allows for two hardware threads to run in one CPU core. By reducing the number of worker threads by 1, we attempt to have one worker thread or constraint thread per hardware thread. So if we have 8 hardware threads, then we have 1 main thread, 7 worker threads, and 1 constraint thread.

If the number of constraints increases, then the number of worker threads should be reduced by the number of hardware threads per CPU core. So we reduced the number of worker threads by 2 when the SUT had a high number of constraints. By doing this we hope that the CPU will assign an entire core to the constraint solving, increasing the overall resources available to the thread. Running our prototype with this setup reduces the runtime in the tested benchmarks.

We switch to the lower number of worker threads if the number of constraints is 10 or higher. However, this number should be adapted to the CPU and the SAT solver used during the generation. The heuristic could also be extended by including other variables. It could be based on the number of constraints, their complexity, the number of parameters, and the levels of the parameters. We leave the generalisation of our technique to future research.

The performance gain obtained by multithreading becomes insignificant with higher amounts of constraints. This is due to the constraints solving becoming the bottleneck of the MCA generation. A multithreaded SAT solver should be used in this case, as this could reduce the impact of the constraint solving. However, we leave the task of adding support for this to future researchers.

7.4 Evaluation

In this section we evaluate the performance of the multithreaded prototype. We compare its performance to the single-threaded prototype, CAgen and ACTS. We use the configuration provided in Section 4.1 to run the benchmarks. ACTS (version 3.2) is run on OpenJDK 11.0.6 using the following command:

```
java -Xms10G -Xmx55G -D doi={strength} -D randstar=off -D output=csv
↪  -D chandler=no -jar acts_3.2.jar path/to/sut.acts
```

CAgen (version 1.1), distributed as a binary file, is run using the following command:

```
time -f "%es" timeout 28800 ./fipo-cli -t {strength} -i
↪  path/to/sut.acts -o results.csv
```

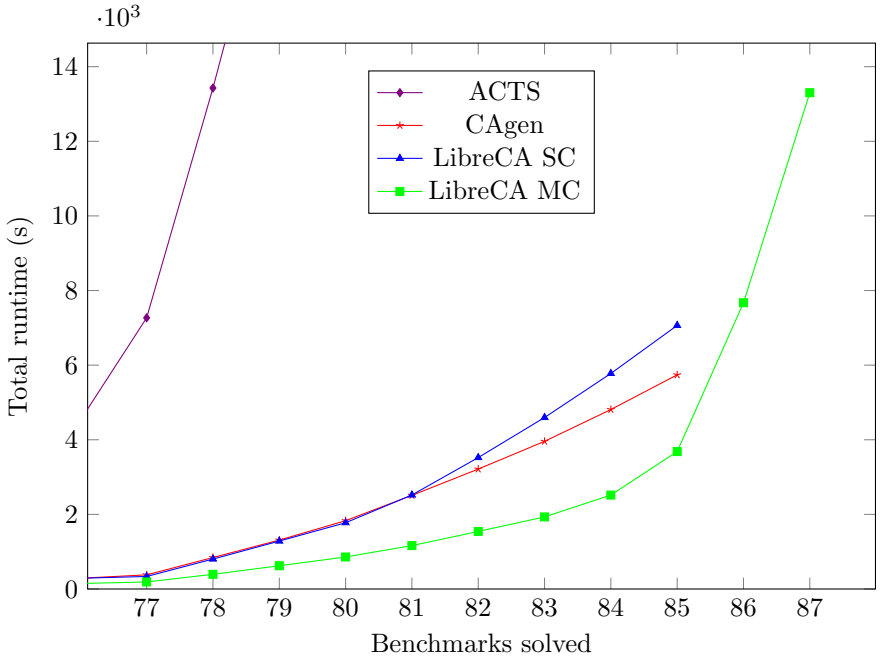
CAgen supports strengths up to $t = 8$, so the benchmarks for higher strengths will be skipped for this tool.

In Figure 7.3, we show the results of benchmarks using SUTs excluding their constraints. We visualise these results in Figure 7.4.a, which is a survival plot. It shows that CAgen takes less time to solve 85 benchmarks than our single-threaded prototype while generating MCAs of similar sizes. However, our multithreaded prototype outperforms all competitors. It manages to solve two additional benchmarks within the timeout of 7200 seconds. The multithreaded prototype shows a geometric mean speedup of 1.46 compared to the single-threaded prototype.

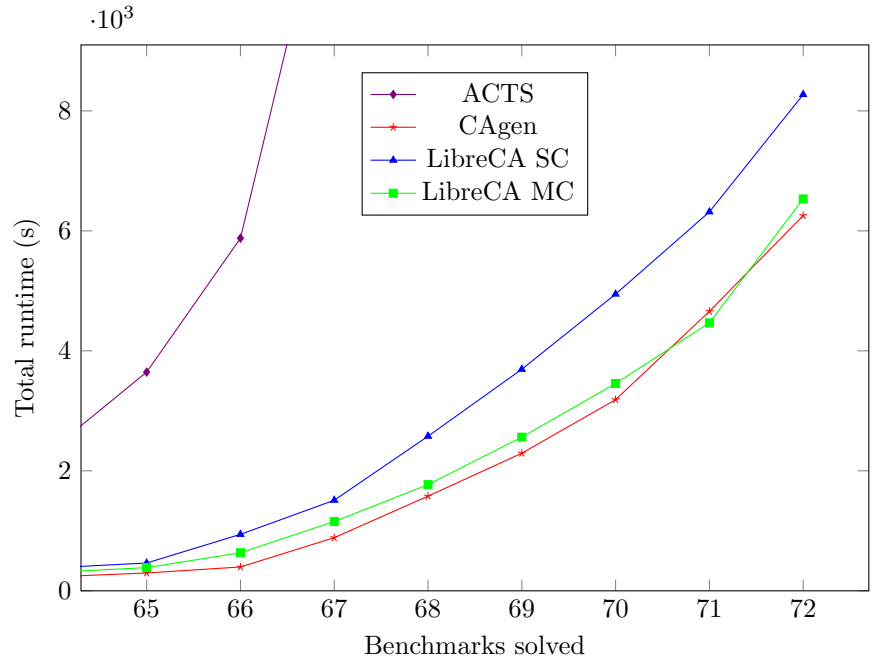
The only benchmark for which the multithreaded prototype is slower than the single-threaded prototype is the 9-way Services. Figure 6.5 shows that this benchmark spends almost all of its time on the VE. Our multithreaded prototype only speeds up the HE, so it will have a reduced effect on benchmarks spending most time on VE. The overhead of the multithreading is most likely the reason why this benchmark is slightly slower.

Figure 7.3: This table contains results of benchmarks using SUTs excluding their constraints. All times are in seconds, and the sizes show the number of tests in the generated MCA. Our prototypes are listed under LIBRECA, where "MC" shows the results of our multithreaded prototype and "SC" the results of our single-threaded prototype. CAGEN is the implementation discussed in [15]. In [3], they discuss ACTS.

			LIBRECA				CAGEN		ACTS	
	Benchmark	t	size	MC	SC	Speedup	size	time	size	time
1	Storage3	9	334901	54	60	1.11	NA		350266	555
2	Xss	10	116195310	69	69	1.00	NA		-	> 28800
3	Processorcomm1	10	694659	189	215	1.14	NA		701529	1154
4	Storage3	10	729108	221	239	1.08	NA		750952	2612
5	Healthcare4	6	39055	205	467	2.28	39018	465	39244	7895
6	Storage5	6	641821	229	482	2.10	644014	467	653540	6159
7	Coveringcerts	7	13486	235	492	2.09	13724	520	13438	10787
8	Insurance	7	13660542	389	743	1.91	13609969	743	13715612	6704
9	Processorcomm1	11	1652141	748	789	1.06	NA		1738591	5553
10	Storage3	11	1476930	940	963	1.02	NA		1511455	9936
11	Healthcare3	7	57302	380	1003	2.64	58185	853	58446	13425
12	Processorcomm2	7	84149	305	1074	3.52	84454	929	86450	11532
13	Services	8	2047454	1166	1180	1.01	1951003	678	2138774	10334
14	Storage4	7	1058772	586	1288	2.20	1049059	705	1054728	9843
15	Processorcomm1	12	3738000	2810	2902	1.03	NA		3788811	18481
16	Storage3	12	2764347	2869	2907	1.01	NA		-	> 28800
17	Services	9	5676211	6690	6608	0.99	NA		-	> 28800
18	Processorcomm2	8	334369	5629	> 7200	NA	-	> 7200	-	> 28800
19	Coveringcerts	8	39639	3989	> 7200	NA	-	> 7200	-	> 28800
	Geometric mean					1.46				



(a) This survival plot visualises Figure 7.3, which tests SUTs ignoring constraints.



(b) This survival plot visualises Figure 7.5, which tests SUTs enforcing their constraints.

Figure 7.4: These survival plots (aka. cactus plot) show the minimum time spend to solve the given number of benchmarks. A benchmarks times out after 7200 seconds. Less time is good, but more tests is even better.

Figure 7.5: This table contains results of benchmarks using SUTs enforcing their constraints. All times are in seconds, and the sizes show the number of tests in the generated MCA. Our prototypes are listed under LIBRECA. "MC" is our multithreaded prototype. "SC" is our single-threaded prototype. CAGEN is the implementation discussed in [15]. In [3], they discuss ACTS.

			LIBRECA				CAGEN		ACTS	
	Benchmark	t	size	MC	SIMPLE	Speedup	size	time	size	time
1	Services	6	163776	79	61	0.77	170626	5	162466	45
2	Storage3	9	49531	66	70	1.07	NA		49824	152
3	Processorcomm1	9	271282	58	73	1.26	NA		278340	354
4	Storage4	6	200370	39	74	1.89	198349	55	201451	725
5	Processorcomm2	6	11514	71	80	1.13	11557	54	11692	525
6	Storage3	10	89500	120	127	1.06	NA		90236	297
7	Storage3	12	213761	158	162	1.02	NA		217824	730
8	Storage3	11	146772	165	169	1.03	NA		148770	500
9	Processorcomm1	10	709211	247	291	1.18	NA		719580	1865
10	Processorcomm1	12	3163597	276	316	1.14	NA		3087411	1256
11	Services	7	482466	520	478	0.92	539893	100	478683	2232
12	Healthcare4	6	37951	248	568	2.29	39018	489	-	> 7200
13	Processorcomm1	11	1539155	846	859	1.02	NA		-	> 7200
14	Processorcomm2	7	40885	895	1067	1.19	40976	893	41430	6510
15	Storage5	6	399313	1010	1116	1.10	399438	1597	-	> 7200
16	Healthcare3	7	53208	616	1252	2.03	51961	1471	-	> 7200
17	Storage4	7	931975	791	1370	1.73	926587	715	-	> 7200
18	Services	8	1011514	2065	1957	0.95	1028149	692	-	> 7200
19	Services	12	1267200	3448	3615	1.05	NA		1267200	3588
20	Services	9	1267200	4020	4091	1.02	NA		-	> 7200
21	Services	11	1267200	5725	6062	1.06	NA		1267200	6007
22	Services	10	1267200	5809	6366	1.10	NA		-	> 7200
	Geometric mean					1.18				

In Figure 7.5, we show the results of benchmarks using SUTs enforcing their constraints. We visualise these results in Figure 7.4.b, which is another survival plot. It shows that CAgen is the fastest tool when constraints are involved while generating MCAs of a similar sizes. The multithreaded prototype performs almost as well as CAgen, but loses in the end. The multithreaded prototype shows a geometric mean speedup of 1.18 compared to the single-threaded prototype. This is lower than the benchmarks ignoring constraints, which means that we should focus on constraint solving in the future if we wish to compete with CAgen.

The benchmarks using the Services SUT are all slow, which is caused by the high number of constraints. The constraint solving is the bottleneck, so all multithreading does is add an overhead. Only at higher strengths does the other work become significant enough compared to the constraint solving. This is why the 9-way and higher are slightly faster, but the lesser strengths are slower.

With these results we can answer our subquestion: *How do we accelerate the IPOG algorithm using multithreading?* By dividing the work of part of the algorithm over multiple threads and removing direct dependencies within the algorithm we can accelerate IPOG using multithreading.

Chapter 8

Acceleration using GPGPU

In this chapter we explore the following question: *How do we accelerate the IPOG algorithm using GPGPU?* We answer this question by creating a GPGPU prototype of IPOG, and compare this to our single-threaded prototype.

Our GPGPU prototype has a rudimentary implementation of the HE. It uses barriers to prevent concurrency related issues. We compare our GPGPU prototype with an equivalent rudimentary single-threaded implementation of the HE on the CPU. Both implementations skip the VE, which results in incomplete MCAs. However, the resulting benchmarks can show whether the use of GPGPU can potentially accelerate the generation of MCAs. Future research could investigate if VE on a GPU is possible. An alternative would be to switch between a GPGPU accelerated HE and a CPU based VE. Switching between GPGPU code and CPU code creates an overhead, so this solution may not be viable.

The results of the benchmarks comparing a GPGPU and a CPU version of the HE are provided in Figure 8.1. From these results we can conclude that our GPGPU prototype is slower than the single-threaded prototype. However, it is important to note that the performance of the GPGPU version is not always slower. Benchmarks taking longer than a minute on the CPU are, on average, faster on the GPU. The geometric mean of the speedup for those benchmarks is 2.24. The most likely reason for this is the barriers present in the GPGPU prototype. Barriers are slow, and take a fixed amount of time to work. When the work per iteration increases, then the impact of the barriers decreases. If the work itself is performed faster on the GPU, then the increase in amount of work would influence the GPGPU version less than the CPU version.

There are three benchmarks for which our prototype failed to produce results, but the single-threaded prototype did. These are the 8-way to 10-way Insurance benchmarks. Unfortunately the GPGPU prototype did not have enough RAM to compute these benchmarks. We could solve this by running the HE on part of the MCA, after which we load the next part of the MCA and continue the computation. The extra time copying parts of the MCA will not create too big an overhead as the transfer bandwidth of a GPU is significant.

			LibreCA		
	Benchmark	t	GPGPU	Base	Speedup
1	Xss	8	215	31	0.15
2	Services	11	252	32	0.13
3	Xss	9	434	37	0.08
4	Services	9	96	37	0.39
5	Services	10	191	47	0.25
6	Storage4	6	32	48	1.53
7	Insurance	6	183	50	0.28
8	Healthcare4	6	64	289	4.48
9	Coveringcerts	7	120	310	2.58
10	Insurance	7	687	373	0.54
11	Processorcomm2	7	90	409	4.53
12	Storage5	6	195	425	2.18
13	Healthcare3	7	302	695	2.30
14	Storage4	7	489	740	1.52
15	Insurance	8	NA	1486	NA
16	Insurance	10	NA	1801	NA
17	Insurance	9	NA	2283	NA
18	Coveringcerts	8	1498	4382	2.93
19	Processorcomm2	8	927	> 7200	NA
	Geometric mean				0.81

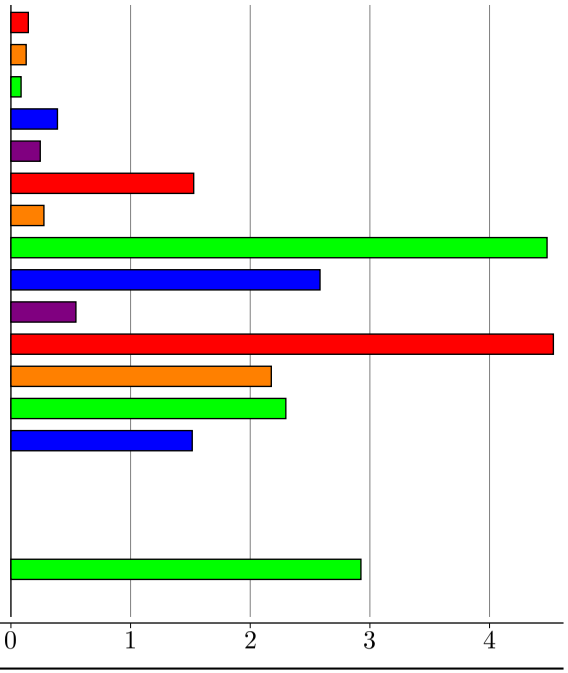


Figure 8.1: This table contains the results of the benchmarks measuring the performance of the GPGPU and single-threaded prototypes. The benchmark was performed on all SUTs listed in Figure A.1 while ignoring the constraints, generating MCAs for the strengths 2 to 12. During the generation of the MCAs VE is skipped. The “Speedup” column represents the relative speedup of the GPGPU prototype compared to the single-threaded prototype. A speedup of 1 indicates equal speed, lower than 1 is slower, and higher than 1 means a higher speed.

We assume that the GPGPU accelerated HE will be faster than the single-threaded version, even when employing this solution.

Section 7.2 shows that concurrency problems can be solved without barriers, so it is possible that the GPGPU version can be adapted in a similar fashion. We leave the implementation of this potential improvement to the future.

With the results shown in Figure 8.1 we can answer the subquestion: *How do we accelerate the IPOG algorithm using GPGPU?* By performing multiple calculations in parallel using GPGPU, we can accelerate part of the algorithm. Removal of direct dependencies within the algorithm have the potential to accelerate IPOG further.

Chapter 9

Conclusion and discussion

9.1 Conclusion

In the introduction we asked ourselves the following research question: *How do we accelerate the IPOG algorithm?* To answer this question we first provided background information about CT, IPOG, and the various technologies we used in Chapters 2, 3 and 4 respectively. In Chapter 5, we provided implementation details of the data structures used in our prototypes.

We attempted to answer the research question using three distinct technologies, which gave us three subquestions. In each of Chapters 6, 7 and 8, we answered one subquestion.

The first subquestion was: *How do we accelerate the IPOG algorithm using bitwise operations?* Chapter 6 introduced novel data structures complementing the existing MCA to increase the efficiency of IPOG. This technique obtains a geometric mean speedup of 1.52 on unconstrained SUTs. The speedup is the result of early on detection of multiple don't-cares, instead of the original single don't-care detection. From this result, we can conclude that acceleration of the IPOG algorithm can be obtained by modifying the implementation details of the algorithm, replacing statements with bitwise operations.

The second subquestion was: *How do we accelerate the IPOG algorithm using multithreading?* We applied multithreading to the HE of the IPOG algorithm. The prototype divides the score calculation between multiple threads. This results in a geometric mean speedup of 1.46 for unconstrained SUTs, and 1.18 for constrained SUTs. From this result, we can conclude that acceleration of the IPOG algorithm can be obtained by creating a multithreaded version of the algorithm, dividing the work between multiple threads.

The third subquestion was: *How do we accelerate the IPOG algorithm using GPGPU?* Finally, we employed GPGPU in an attempt to accelerate the IPOG algorithm. We did not manage to increase the speed of the HE. However, there are cases where the GPGPU prototype outperformed the single-threaded prototype. It is likely that the GPGPU prototype can be improved upon using the techniques employed in the multithreading prototype. From the results so

far, we can conclude that acceleration of the IPOG algorithm can potentially be obtained by creating a GPGPU version of the algorithm, dividing the work over multiple hardware threads.

Answering the three subquestions allows us to answer our research question. We have proven that we can accelerate IPOG using bitwise operations and multithreading. GPGPU shows potential when it comes to acceleration of IPOG. The application of the techniques in our prototypes was a time consuming process, and specific to IPOG. Using these techniques on other algorithms will require a similar investment, and there is no guarantee that the technique will increase performance. This report shows how to accelerate IPOG, but this know-how will have limited use when accelerating other algorithms.

9.2 Future work

In this section we will discuss various improvements on our implementations, and how future research could improve the results obtained. In previous chapters we presented various potential improvements regarding LibreCA, which we summarise in Section 9.2.1. The next section, Section 9.2.2, will discuss future research based on related work.

9.2.1 Based on this report

In this section we will list the proposed future research mentioned throughout this report in order of occurrence.

In Section 5.1.6, we mention that during the run of IPOG we will repeatedly iterate through all PCs. Currently we create a list of all PCs and save them in memory. However, calculating the next PC is inexpensive, and increasing the index is also simple. Further research could find out whether keeping a list or calculating PCs in place is faster.

At the end of Section 5.2, we suggested that we could recompile our implementation for a given SUT once the prototype is called. This would mean that we compile an implementation on demand for each SUT. For complex SUTs, the overhead caused by the compilation will probably be less than the time gained by the compile time optimisations.

In Chapter 6, we describe an optimisation of the IPOG algorithm. Further research will be needed to find if this technique is also effective in the GPGPU prototype. In the same chapter we mention that the performance of the VE can further be increased if the iteration order of PCs and rows in the MCA is changed. However, future researchers should reevaluate other design decisions when attempting this alternate implementation, as most decisions will be impacted by this reordering.

Our multithreading prototype only divides work between threads if enough work is present, as stated in Section 4.3. Currently the heuristic to determine

when to switch is the amount of PCs. Furthermore, Section 7.3 mentions another heuristic is used to determine the number of worker threads. Future research could provide alternatives to these heuristics and evaluate their impact on the overall performance of the prototype. These heuristics could include the number of constraints, their complexity, the number of parameters, and the levels of the parameters.

In Section 7.4, we notice that the constraint solving support in our prototypes is slower than the one in CAgen. CAgen uses a method called "forbidden tuples" to solve constraints [32]. There is also a GPGPU SAT solver called CUD@SAT, introduced in [28]. Future research could add support for these solvers to our prototypes.

Currently, we have yet to create a multithreading and/or GPGPU version of VE, as mentioned in Chapters 7 and 8. By dividing the work performed during VE between threads we can potentially improve the overall performance of our multithreaded and GPGPU prototypes.

The GPGPU prototype does not yet implement the techniques used in the multithreading solution, as explained in Chapter 8. The performance of the GPGPU prototype could greatly increase if these techniques could be translated, as it would remove the number of barriers and other types of communication in the implementation.

9.2.2 Based on related work

The authors of [20, 38] introduce IPO, which can create CAs for SUTs of which the parameters have a fixed number of values. IPOG then extended this algorithm to generate MCAs instead, in [21]. This report further extends this algorithm, following the implementation details provided by Kleine et al. in both [14, 15]. In [15], they introduce the implementation details of data structures which are the basis for the data structures compared in Chapter 5. The same paper also introduces techniques to improve the performance of both the HE and VE of IPOG. In [14], they compare various heuristics to be used during HE when selecting new values. This paper also investigates how the ordering of parameters affects the resulting MCA. It is yet unknown if the various improvements mentioned by Kleine et al. and the various techniques used in this report are beneficial when combined. Future research could eliminate these unknowns.

In [37] the authors describe an alternate HE, where the best value in all rows is selected. Remember that the original IPOG chooses the best value for one row at a time. This version produces smaller MCAs, but it also takes more time to generate said MCA. However, the extra work has no dependency on other decisions, which makes it perfect to accelerate using multithreading or GPGPU. Which is why they implemented a multithreaded prototype. Their results show promising MCA sizes, but their performance is lacking. As future research, we could implement this alternate HE using bitwise operations and multithreading, and evaluate its performance.

Duan et al. [8, 9] introduce a new VE implementation based on “Minimum Vertex Coloring” problem to minimise the number of rows added during VE. They create a graph model capturing the conflict relationship between interactions and tests, and then solve the classical NP-hard “Minimum Vertex Coloring” problem on this graph. This solution provides better MCA sizes at the cost of additional runtime. Their insight on the importance and properties of don’t-cares in MCAs served as the basis of our bitwise prototype, as is explained in Chapter 6. As future research, we could investigate if the VE alternative proposed in [8, 9] could benefit from the same acceleration. This VE uses an optimization technique to minimize generated array using graph coloring.

The techniques introduced in this report can possibly be applied to other greedy algorithms. Besides IPOG, there is One-Test-At-A-Time (OTAT), which is discussed in [7, 35]. As the name suggests, OTAT builds MCAs one test case at a time, choosing the locally optimal solution. This algorithm uses more memory due to the larger CM. Our proposed techniques may be relevant to these greedy algorithms for MCA generation.

Acronyms

ACTS Automated Combinatorial Testing for Software. 11, 24, 65

ALU Arithmetic Logic Unit. 28, 29

API Application Programming Interface. 34

CA Covering Array. 24, 77

CAgen Covering Array Generator [*sic*]. 11, 24, 65, 69, 77

CHıP Configurable Hybrid Parallel Covering Array Constructor. 10

CM Coverage-map. 7–9, 14–22, 35, 36, 39, 43, 48, 49, 61, 63, 64, 78

CPU Central Processing Unit. 3, 5–7, 14–22, 28–30, 35–39, 43, 45, 64, 65, 71

CT Combinatorial Testing. 1, 5, 6, 8, 10, 75

CUDA Compute Unified Device Architecture. 25, 27, 28, 30–34

GPGPU General-Purpose computing on Graphics Processing Units. i, 2, 3, 25, 27, 28, 30, 32, 48, 52, 71–73, 75–77

GPU Graphics Processing Unit. 2, 3, 27–32, 34, 43, 45, 71

HE Horizontal Extension. 14, 16–20, 23, 24, 27, 44, 47–49, 53, 55–57, 61, 65, 71, 73, 75, 77

IP Internet Protocol. 19

IPO In-Parameter-Order. 24, 77

IPOG In-Parameter-Order-General. i, 2, 3, 10, 13–15, 17–19, 21–24, 26, 27, 35–37, 39, 41–43, 47, 48, 59, 61, 62, 69, 71, 73, 75–78

MCA Mixed-level Covering Array. i, 1–3, 5–11, 13, 14, 16–26, 35, 36, 41–45, 47, 48, 53, 54, 57, 58, 61–66, 68, 69, 71, 72, 75–78

OpenCL Open Compute Language. 28, 30–32, 34

OS Operating System. 1, 5–7, 14–22, 35, 36

OTAT One-Test-At-A-Time. 78

PC Parameter Combination. 6, 20–22, 26, 27, 35–38, 41–43, 47–49, 51, 57, 62, 76, 77

PICT Pairwise Independent Combinatorial Testing. 10

RAM Random-access memory. 22, 71

SA Simulated Annealing. 9–11

SAT Boolean satisfiability problem. 9–11, 25, 42, 65, 77

SUT System Under Test. i, 1, 5–9, 13, 14, 16, 18–25, 40–45, 54, 56–58, 61, 64–69, 72, 75–77

VE Vertical Extension. 14, 16–20, 23, 24, 27, 43, 44, 47, 53–57, 62, 65, 71, 72, 76–78

Bibliography

[1] B. S. Ahmed et al. “Constrained Interaction Testing: A Systematic Literature Study”. In: *IEEE Access* 5 (2017), pp. 25706–25730. DOI: 10.1109/ACCESS.2017.2771562.

[2] D. Blue et al. “Interaction-based Test-suite Minimization”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 182–191. DOI: 10.1109/ICSE.2013.6606564.

[3] M. N. Borazjany et al. “Combinatorial Testing of ACTS: A Case Study”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 2012, pp. 591–600. DOI: 10.1109/ICST.2012.146.

[4] Pierre Collet. “Why GPGPUs for Evolutionary Computation?” In: *Massively Parallel Evolutionary Computation on GPGPUs*. Ed. by Shigeyoshi Tsutsui and Pierre Collet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 3–14. DOI: 10.1007/978-3-642-37959-8_1.

[5] Microsoft Corp. *PICT*. URL: http://github.com/microsoft/pict (visited on 2021-09-01).

[6] Jacek Czerwonka. *Pairwise Testing*. URL: https://jaccz.github.io/pairwise/ (visited on 2021-09-01).

[7] Jacek Czerwonka. “Pairwise testing in the real world: Practical extensions to test-case scenarios”. In: *24th Pacific Northwest Software Quality Conference*. 2006, pp. 419–430. URL: http://www.uploads.pnsqc.org/proceedings/pnsqc2006.pdf (visited on 2021-09-01).

[8] F. Duan et al. “Optimizing IPOG’s Vertical Growth with Constraints Based on Hypergraph Coloring”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2017, pp. 181–188. DOI: 10.1109/ICSTW.2017.37.

[9] Feng Duan et al. “Improving IPOG’s vertical growth based on a graph coloring scheme”. In: *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 1–8. DOI: 10.1109/ICSTW.2015.7107444.

[10] Adrian R. G. Harwood. “GPU-powered, interactive flow simulation on a peer-to-peer group of mobile devices”. In: *Advances in Engineering Software* 133 (2019), pp. 39–51. DOI: 10.1016/j.advengsoft.2019.04.003.

[11] Guangyong He Liqiang and Zhang. “Parallel Branch Prediction on GPU Platform”. In: *High Performance Computing and Applications*. Ed. by Wu Zhang et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 153–160. DOI: 10.1007/978-3-642-11842-5_20.

[12] Linghuan Hu et al. “How does combinatorial testing perform in the real world: an empirical study”. In: *Empirical Software Engineering* 25.4 (2020-07), pp. 2661–2693. DOI: 10.1007/s10664-019-09799-2.

[13] Raju K and Niranjan N. Chiplunkar. “A survey on techniques for cooperative CPU-GPU computing”. In: *Sustainable Computing: Informatics and Systems* 19 (2018), pp. 72–85. DOI: 10.1016/j.suscom.2018.07.010.

[14] Kristoffer Kleine, Ilias Kotsireas, and Dimitris E. Simos. “Evaluation of Tie-Breaking and Parameter Ordering for the IPO Family of Algorithms Used in Covering Array Generation”. In: *Combinatorial Algorithms*. Ed. by Costas Iliopoulos, Hon Wai Leong, and Wing-Kin Sung. Cham: Springer International Publishing, 2018, pp. 189–200. DOI: 10.1007/978-3-319-94667-2_16.

[15] Kristoffer Kleine and Dimitris E. Simos. “An Efficient Design and Implementation of the In-Parameter-Order Algorithm”. In: *Mathematics in Computer Science* 12.1 (2018-03), pp. 51–67. DOI: 10.1007/s11786-017-0326-0.

[16] P. Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019-05, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[17] Kronos Group. *Conformant Products*. URL: https://www.khronos.org/conformance/adopters/conformant-products/opencl (visited on 2021-09-01).

[18] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. “Software fault interactions and implications for software testing”. In: *IEEE Transactions on Software Engineering* 30.6 (2004-06), pp. 418–421. DOI: 10.1109/TSE.2004.24.

[19] R. Kuhn et al. “Combinatorial Software Testing”. In: *Computer* 42.8 (2009-08), 94–96. DOI: 10.1109/MC.2009.253.

[20] Kuo-Chung Tai and Yu Lei. “A test generation strategy for pairwise testing”. In: *IEEE Transactions on Software Engineering* 28.1 (2002-01), pp. 109–111. DOI: 10.1109/32.979992.

[21] Y. Lei et al. “IPOG: A General Strategy for T-Way Software Testing”. In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. 2007, pp. 549–556. DOI: 10.1109/ECBS.2007.47.

[22] S. Liu, C. Eisenbeis, and J. Gaudiot. “Speculative Execution on GPU: An Exploratory Study”. In: *2010 39th International Conference on Parallel Processing*. 2010, pp. 453–461. DOI: 10.1109/ICPP.2010.53.

[23] Ogier Maitre. “Understanding NVIDIA GPGPU Hardware”. In: *Massively Parallel Evolutionary Computation on GPGPUs*. Ed. by Shigeyoshi Tsutsui and Pierre Collet. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 15–34. DOI: 10.1007/978-3-642-37959-8_2.

[24] H. Mercan, C. Yilmaz, and K. Kaya. “CHiP: A Configurable Hybrid Parallel Covering Array Constructor”. In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. DOI: 10.1109/TSE.2018.2837759.

[25] Aaftab Munshi. *The OpenCL Specification*. Version 19. Khronos OpenCL Working Group. 2012-11. URL: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf.

[26] NVIDIA Corporation. *CUDA GPUs*. URL: https://developer.nvidia.com/cuda-gpus (visited on 2021-09-01).

[27] NVIDIA Corporation. *CUDA Toolkit Documentation*. 2019-11. URL: https://docs.nvidia.com/cuda/ (visited on 2021-09-01).

[28] Alessandro Dal Palù et al. “CUD@SAT: SAT solving on GPUs”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 27.3 (2015), pp. 293–316. DOI: 10.1080/0952813X.2014.954274.

[29] D. E. Simos et al. “Practical Combinatorial Testing for XSS Detection using Locally Optimized Attack Models”. In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2019, pp. 122–130. DOI: 10.1109/ICSTW.2019.00040.

[30] The Rust Team. *Official website of Rust*. URL: https://www.rust-lang.org/ (visited on 2021-09-01).

[31] Wei-Tek Tsai and Guanqiu Qi. “Introduction”. In: *Combinatorial Testing in Cloud Computing*. Springer Singapore, 2017, pp. 1–13. DOI: 10.1007/978-981-10-4481-6_1.

[32] M. Wagner et al. “CAGEN: A fast combinatorial test generation tool with support for constraints and higher-index arrays”. In: *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2020, pp. 191–200. DOI: 10.1109/ICSTW50294.2020.00041.

[33] P. Wojciak and R. Tzoref-Brill. “System Level Combinatorial Testing in Practice – The Concurrent Maintenance Case Study”. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 2014, pp. 103–112. DOI: 10.1109/ICST.2014.23.

[34] A. Yamada et al. “Optimization of Combinatorial Testing by Incremental SAT Solving”. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. 2015-04, pp. 1–10. DOI: 10.1109/ICST.2015.7102599.

[35] Akihisa Yamada et al. “Greedy combinatorial test case generation using unsatisfiable cores”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 614–624. DOI: 10.1145/2970276.2970335.

[36] Jun Yan and Jian Zhang. “A Backtracking Search Tool for Constructing Combinatorial Test Suites”. In: *J. Syst. Softw.* 81.10 (2008-10), pp. 1681–1693. DOI: 10.1016/j.jss.2008.02.034.

[37] Mohammed I. Younis and Kamal Z. Zamli. “MC-MIPOG: A Parallel t-Way Test Generation Strategy for Multicore Systems”. In: *ETRI Journal* 32.1 (2010), pp. 73–83. DOI: 10.4218/etrij.10.0109.0266.

[38] Yu Lei and K. C. Tai. “In-parameter-order: a test generation strategy for pairwise testing”. In: *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*. 1998, pp. 254–261. DOI: 10.1109/HASE.1998.731623.

Appendix A

Benchmark info

	Benchmark	$\|P\|$	Levels	$\|\phi\|$
1	Banking1	5	$4^1 3^4$	112
2	Banking2	15	$4^1 2^{14}$	3
3	Commprotocol	11	$7^1 2^{10}$	128
4	Concurrency	5	2^5	7
5	Coveringcerts	33	$5^1 4^1 3^8 2^{23}$	0
6	Healthcare1	10	$6^1 5^1 3^2 2^6$	21
7	Healthcare2	12	$4^1 3^6 2^5$	25
8	Healthcare3	29	$6^1 5^1 4^5 3^6 2^{16}$	31
9	Healthcare4	35	$7^1 6^1 5^2 4^6 3^{12} 2^{13}$	22
10	Insurance	14	$31^1 17^1 13^1 11^1 6^2 5^1 3^1 2^6$	0
11	Networkmgmt	9	$11^1 10^2 5^3 4^1 2^2$	20
12	Processorcomm1	15	$4^6 3^6 2^3$	13
13	Processorcomm2	25	$5^2 4^8 3^{12} 2^3$	125
14	Services	13	$10^2 8^2 5^2 3^4 2^3$	388
15	Storage1	4	$5^1 4^1 3^1 2^1$	95
16	Storage2	5	$6^1 3^4$	0
17	Storage3	15	$8^1 6^1 5^3 3^1 2^9$	48
18	Storage4	20	$13^1 10^1 7^1 6^2 5^2 4^1 3^7 2^5$	24
19	Storage5	23	$11^1 10^2 9^1 8^1 6^2 5^3 3^8 2^5$	151
20	Systemmgmt	10	$5^1 3^4 2^5$	17
21	Telecom	10	$6^1 5^1 4^2 3^1 2^5$	21
22	Xss	11	$23^1 15^1 14^1 11^1 9^1 3^6$	0

Figure A.1: Benchmark information. The sizes of benchmarks are expressed in the form $g_1^{k_1} g_2^{k_2} \cdots g_n^{k_n}$ which means that for each i there are k_i parameters that have g_i values.