GPU-BASED PARALLEL COMPUTING METHODS FOR CONSTRUCTING COVERING ARRAYS

by

Hanefi Mercan

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

August 2015

GPU-BASED PARALLEL COMPUTING METHODS FOR CONSTRUCTING COVERING ARRAYS

Approved by:

Assist. Prof. Dr. Cemal Yılmaz .....
(Thesis Supervisor)

Assist. Prof. Dr. Kamer Kaya .....
(Thesis Co-Supervisor)

Assoc. Prof. Dr. Hüsnü Yenigün .....

Prof. Dr. Bülent Çatay .....

Assist. Prof. Dr. Hasan Sözer .....

Date of Approval: 03/08/2015

GPU-BASED PARALLEL COMPUTING METHODS FOR CONSTRUCTING COVERING ARRAYS

Hanefi Mercan

Computer Science and Engineering, MS Thesis, 2015

Thesis Supervisor: Asst. Prof. Cemal Yılmaz

Keywords: Combinatorial interaction testing, covering array, simulated annealing, cuda, parallel computing, combinatorial coverage measurement

Abstract

As software systems becomes more complex, demand for efficient approaches to test these kind of systems with a lower cost is increased highly, too. One example of such applications can be given highly configurable software systems such as web servers (e.g. Apache) and databases (e.g. MySQL). They have many configurable options which interact each other and these option interactions lead having exponential growth of option configurations. Hence, these software systems become more prone to bugs which are caused by the interaction of options.

A solution to this problem can be combinatorial interaction testing which systematically samples the configuration space and tests each of these samples, individually. Combinatorial interaction testing computes a small set of option configurations to be used as test suites, called covering arrays. A *t-way covering array* aims to cover all t-length option

interactions of system under test with a minimum number of configurations where t is a small number in practical cases. Applications of covering arrays are especially encouraged after many researches empirically pointed out that substantial number of faults are caused by smaller value of option interaction.

Nevertheless, computing covering arrays with a minimal number of configurations in a reasonable time is not easy task, especially when the configuration space is large and system has inter-option constraints that invalidate some configurations. Therefore, this study field attracts various researchers. Although most of approaches suffer in scalability issue, many successful attempts have been also done to construct covering arrays. However, as the configuration scape gets larger, most of the approaches start to suffer.

Combinatorial problems e.g., in our case constructing covering arrays, are mainly solved by using efficient counting techniques. Based on this assumption, we conjecture that covering arrays can be computed using parallel algorithms efficiently since counting is an easy task which can be carried out with parallel programming strategies. Although different architectures are effective in different researches, we choose to use GPU-based parallel computing techniques since GPUs have hundreds even sometimes thousands of cores however with small arithmetic logic units. Despite the fact that these cores are exceptionally constrained and limited, they serve our purpose very well since all we need to do is basic counting, repeatedly. We apply this idea in order to decrease the computation time on a meta-heuristic, search method simulated annealing, which is well studied in construction of covering arrays and, in general, gives the smallest size results in previous studies. Moreover, we present a technique to generate multiple neighbour states in each step of simulated annealing in parallel. Finally, we propose a novel hybrid approach using SAT solver with parallel computing techniques to decrease the negative effect of pure random search and decrease the covering array size further. Our results prove our assumption that parallel computing is an effective and efficient way to compute combinatorial objects.

GPU TABANLI PARALEL HESAPLAMA YÖNTEMLERİ İLE KAPSAYAN DİZİLER OLUŞTURMA

Hanefi Mercan

Bilgisayar Bilimleri ve Mühendisliği, Yükseklisans Tezi, 2015

Tez Danışmanı: Yar. Doç. Cemal Yılmaz

Özet

Yazılım sistemleri daha karmaşık hale geldikçe, bu tip sistemleri düşük maliyetli test etmek için etkili tekniklere olan talep de artmaktadır. Bunlara örnek olarak web sunucuları (Apache vb.) ve veritabanları (MsSQL vb.) gibi yapılandırılabilirliği yüksek yazılım sistemleri verilebilir. Bu sistemler birbiriyle etkileşim içinde olan birçok yapılandırabilir parametrelere sahiptir ve bu etkileşimler üstel büyüme hızıyla parametre konfigürasyonlarının sayısının artmasına yol açar. Bundan dolayı, bu tip yazılım sistemleri parametrelerin etkileşimlerinden dolayı oluşabilecek hatalara karşı daha çok eğilimlidir.

Bu soruna bir çözüm olarak konfigürasyon uzayını sistematik şekilde kümeleyip ve bu kümeleri ayrı ayrı test eden kombinatoryal etkileşim testi (combinatorial interaction testing) verilebilir. Kombinatoryal etkileşim testi, az sayıda parametre konfigürasyonları içeren test senaryoları olarak kullanılması için kapsayan diziler adı verilen objeleri üretir. Bir *t-yollu kapsayan dizisi* (t-way covering array) test edilecek sistemin bütün t-yollu parametrelerinin değer kombinasyonlarını en küçük sayıda konfigürasyon kullanarak kapsamayı hedefler. Yapılan birçok araştırmanın kayda değer çoklukta hataların küçük sayıda parametre etkileşimlerinden kaynaklandığını göstermesinin ardından, kapsayan dizinin uygulamalarına özellikle teşvik edilmiştir.

Yine de, özellikle de konfigürasyon uzayı büyük olduğunda ve sistem içinde parametreler arasında bazı konfigürasyonları geçersiz kılan kısıtlamalar olduğunda, minimum sayıda konfigürasyon içeren kapsayan diziler oluşturmak kolay bir iş değildir. Bundan ötürü, bu çalışma alanı farklı alandan birçok araştırmacıların ilgisini çekmektedir. Çoğu çalışma ölçeklendirme konusunda sorun yaşamasına rağmen, kapsayan dizi oluşturma konusunda bazı başarılı çalışmalarda yapılmıştır. Fakat, konfigürasyon uzayı büyüdükçe, çoğu yaklaşım zorlanmaya başlar.

Kombinatorik problemler, bizim durumda kapsayan dizi oluşturmak, çoğunlukla etkili sayma teknikleri kullanılarak çözülür. Bu varsayımı baz alarak, sayma problemlerinin kolay bir iş olmasından ve paralel programlama teknikleri kullanılarak verimli bir şekilde çözülebileceğinden dolayı, kapsayan dizilerin paralel algoritmalar kullanılarak etkili bir şekilde oluşturulabileceğine dair öngörüde bulunuyoruz. Farklı mimariler farklı araştırma alanlarında daha etkili olabileceğinden dolayı, biz GPU tabanlı paralel programlama teknikleri kullanmaya karar verdik. Çünkü GPU'ların aritmetik hesaplama birimleri küçük olmasına karşın, yüzlerce çekirdekleri, hatta bazen binlerce çekirdekleri olabilir. Bu çekirdeklerin kapasiteleri kısıtlı ve sınırlı olmalarına rağmen, bizim tek yapmak istediğimiz defalarca basit sayma işlemleri olduğu için bizim çalışmamızda amacımıza çok iyi hizmet ederler. Bu fikrimizi hesaplama zamanını azaltmak için daha önce birçok defa kapsayan dizi oluşturmada kullanılmış ve çoğu zaman en küçük boyutlarda sonuçlar vermiş olan benzetilmiş tavlama algoritması (simulated annealing) üzerinde uyguladık. Bunlara ek olarak, benzetilmiş tavlama algoritmasının her adımında paralel olarak çoklu sayıda komşu durumları üretebilen bir teknik geliştirdik. Son olarak da, uzayı tamamen rastgele aramanın kötü etkisini düşürmek ve kapsayan dizilerin boyutunu daha da azaltmak için SAT (SATisfiability) algoritması ve paralel programlama teknikleri kullanarak melez bir yaklaşım öne sürdük.

To my Family

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Cemal Yılmaz whose expertise, understanding, and patience, added considerably to my graduate experience. During my master education, he gave me the moral support and the freedom I needed to move on. It has been great honor to work under his guidance.

My co-advisor, Prof. Kamer Kaya, was always there to listen and give advice. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work. I also would like to thank the other members of my thesis committee, Prof. Hüsnü Yenigün, Prof. Bülent Çatay, and Prof. Hasan Sözer for their insightful comments.

A special thanks goes to Gülşen Demiröz for her great research cooperation in thesis project, insightful comments and support of my master study. I am indebted to my labmates: Arsalan Javeed, Rahim Dehkharghani, and Uğur Koç for the inspiring discussions, and research cooperations. Also, I would like to convey very special thanks to my friends Ercan Kalalı, Zhenishbek Zhakypov, Erkan Duman and Kağan Aksoydan for all the fun we have had.

Moreover, I would like to acknowledge Sabancı Universtiy and Scientific Technological Research Council of Turkey (TUBITAK) for supporting me throughout my graduate education.

Most importantly, none of this would have been possible without the love and patience of my parents Hatice Mercan and Ahmet Remzi Mercan, my sisters and brother Neslihan Doğan, Perihan Mercan and Faruk Mercan for supporting spiritually me throughout my life. Lastly, my heartiest thanks to my dear fiancée Sevinç Göğebakan for being in my life and empowering me with her great love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

M	System model
O	Set of system options
V	Set of option settings
Q	Set of system-wide inter-option constraints
k	Number of options
N	Size of covering array
t	Strength of covering array
R	Set of t-tuples
s_{ij}	option-value pair
S	State in an inner search process
S_u	Upper boundary state
S_l	Lower boundary state
B	Number of blocks in a grid
T	Number of threads in a block
w	Warp
N_u	Number of uncovered t-tuples

LIST OF ABBREVIATIONS

CS	Computer Science.
SUT	System Under Test.
CIT	Combinatorial Interaction Testing.
CA	Covering Array.
GPU	Graphics Processing Unit.
SA	Simulated Annealing.
NS	Neighbour State.
CCM	Combinatorial Coverage Measurement.
CUDA	Compute Unified Device Architecture.
SAT	Boolean satisfiability testing.
CNF	Conjunctive Normal Form.
ACTS	Advanced Combinatorial Testing System.
NIST	National Institute Standards and Technology.
IS	Initial State.
SA	Hamming Distance.

1

INTRODUCTION

Software testing plays an important role in software development cycle. It helps to produce more reliable systems and improves the quality. Defects and errors are identified and located in the testing phase so that they can be fixed before the product is released. Therefore, the testing part aims to eliminate the inconsistencies in the software development process.

In testing phase, getting a full coverage of the System Under Test (SUT) needs to be satisfied if one desires to identify and locate the all the existing defects i.e., all possible scenarios (requirements) of the system behaviours needs to be included in the test cases. However, testing all possible scenarios (exhaustive testing) may not be feasible or affordable most of the time. One example of such applications can be highly configurable software systems such as web servers (e.g. Apache) and databases (e.g. MySQL). They have many configurable options which interact with each other. These option interactions lead to having exponential growth of possible configurations. Hence, these software systems become more prone to bugs which are caused by the interaction of options. For example, a software system having 50 options with binary settings (values) may lead to having 2^{50} different configurations. Therefore, a full coverage of all possible configurations is not feasible in general, even if exhaustive testing is desirable.

One solution for this problem can be Combinatorial Interaction Testing (CIT) [59] which is used widely to test software systems. CIT takes a configuration space model of SUT as an input which includes a set of configurable options, their possible settings, and a set of system-wide inter-option constraints that invalidate some configurations. Then, CIT samples the configuration space based on a coverage criteria and tests each of these samples individually.

An important part of CIT is to generate a test suite in a way that it both contains a small number of configurations and covers all requirements without violating any constraints if there exists any. In CIT, mostly, Covering Arrays are used as test suites. A *t-way Covering Array* (CA) is a mathematical object which has N (size) rows and k (number of options) columns ensuring every *t-tuple* (length of t) is covered by some row at least once where t is called the strength of the CA. Each column of CA keeps the corresponding option settings and each row is referred as a configuration option where the test case is executed on (or test cases [57]).

In general, main goal of CA is to get full coverage based on some criteria so that every desired requirement is satisfied. Once t-way CA is constructed, every test case is executed by configuring the option values of SUT as suggested by the configurations of CA. Therefore, it is important that keeping CA construction time shorter to start testing earlier and keeping CA size smaller to finish testing sooner (under certain assumptions). CAs are of great practical importance as also apparent from more than 50 papers published only for construction of CAs [41].

CAs have been extensively used for configuration testing, product line testing, systematic testing of multi-threaded applications, input parameter testing, etc. [4, 20, 34, 37, 44, 58]. In these researches, many empirical results suggest that most of the failures are occurred by a small number of option interaction. Therefore, a t-way CA where t is a small number ($2 \leq t \leq 6$) becomes an efficient test suite in identifying and locating bugs with small number of configurations.

Various methods have been proposed for constructing CAs in a smaller size and a reasonable time as Nie et al suggested [41]. However, some of them suffers as the configuration space gets larger since it affects the number of t-tuples exponentially. Having large number of t-tuples may make the problem even harder. Furthermore, in practical scenarios,

not all t-tuples are valid i.e., there may be system-wide constraints between some option settings which make constructing a CA even harder without violating any of them. Despite these facts, we believe that this problem is indeed a simple counting task and if the objects can be counted in an efficient way, and a great improvement may be achieved both in time and size. Considering the suitability of counting for parallelization, we claim that such combinatorial problems can be solved with parallel computing techniques more effectively. As different architectures can be more suitable for different methods, in this case, we choose to move forward with Graphics Processing Unit (GPU). Modern GPUs have thousands of cores however with a relatively less powerful arithmetic logic unit. This speciality of GPUs motivates us to employ them since all we need to do is a simple task of counting but as possible as concurrently.

Simulated Annealing (SA) is a metaheuristic search algorithm which is used for CA construction very often [11, 15, 50, 51]. Even though it is a local search algorithm, its probabilistic decision function saves us getting trapped in a local minima, so that, a smaller size CA can be constructed. SA consists of 2 main steps; generating a Neighbour State (NS) and calculating the gain (fitness function) of accepting NS. NS is simply generated by changing a part of current state and the gain measures how good the NS is compared to current state. In our case, the NS is same as current state with only one change in one of the option value. On the other hand, the gain is the difference between the number of t-tuples covered by the NS and current state. The main issue in SA is that one may need to repeat these steps for thousands of times or even sometimes millions of times to obtain better solutions. Therefore, counting the number of t-tuples in a shorter time carries big importance in SA.

For the reasons that we mention above, we propose parallel methods for SA in order to construct CAs in a reasonable time and with a smaller number of configurations compared to the existing approaches. Moreover, we also give an approach to generate multiple neighbour states in parallel. Finally, we combine all the described methods and propose a novel hybrid approach to construct CAs using a SAT solver.

Our contributions can be summarized as follows: (1) We give an efficient algorithm to measure combinatorial coverage of a CA. (2) We present a novel parallel algorithm for fitness function computing technique without enumerating all possible t-tuples. (3) Several parallel computing techniques are described to increase the efficiency both in time and size. (4) A novel hybrid approach is proposed for faster convergence and better quality, especially for the large and dense constraint spaces.

The rest of the thesis is structured as follows: in Chapter 2, a brief background information is given. Chapter 3 points out the strengths and weaknesses of the existing methods and tools which are used in CIT. Both sequential and parallel approaches for constructing CAs are explained in Chapter 4 in detail. We present our experimental results and analysis in Chapter 5. Chapter 6 states the concluding remarks and ideas for the future work.

2

BACKGROUND

In this chapter, we give background information about Combinatorial Interaction Testing, Covering Arrays, Simulated Annealing, CUDA Parallel Programming Platform and Satisfiability Problem.

2.1. Combinatorial Interaction Testing

Combinatorial Interaction Testing (CIT) is widely used to sample program inputs and, also to test highly configurable software systems, multithreaded applications, Graphical User Interface (GUI) applications, etc. Main goal of the CIT is to identify the faults which are triggered by the interaction of options. Yilmaz et al. [59] argue that CIT can be divided into four phases as shown in Figure 2.1.

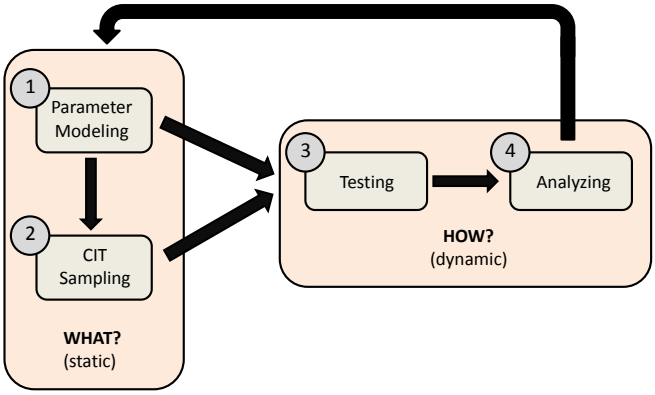

Figure 2.1: Four phases of CIT

The first phase is to model the characteristics of the System Under Test (SUT) such as inputs, configurations, and sequences of operations. The second phase is to sample the requirements of the model which are generated in the first phase to cover all the expectations from testing, e.g., two pair of all options. The third and fourth phases are testing and analysing. In testing phase, generated test cases from sampling phase are tested either in a batch mode, incrementally, or adaptively. As last, the test results are examined and causes of failures are revealed in the analysing phase.

In this thesis, we are interested in the second phase of CIT which is generating the test suites with the given requirements. System specifications are fed into our algorithm and we construct test suites, i.e., Covering Arrays.

2.2. Covering Arrays

In every software testing strategy, there are some requirements which has to be covered with respect to characteristics of SUT in order to reveal the bugs. In our system model $M =< O, V, Q >$ $O = \{o_1, o_2, \ldots, o_k\}$ stands for the options (factors) of SUT and V_i's $\in V = \{V_1, V_2, \ldots, V_k\}$ are the corresponding settings (values) sets for each option o_i, where $1 \leq i \leq k$. Additionally, $Q = \{q_1, q_2, \ldots, q_m\}$ is the set of system-wide inter-option constraints if there exist any for the SUT.

In CIT, the requirements are the t-tuple set $R = \{R_1, R_2, ..., R_n\}$, where each t-tuple $R_i = \{s_{ij_1}, s_{ij_2}, \ldots, s_{ij_t}\}$ has t distinct option-values $s_{ij} =< o_i, v_j >$ pairs, where $v_j \in V_i$ and $1 \leq t \leq n$.

Definition 1 *A t-tuple $R_i = \{s_{ij_1}, s_{ij_2}, ..., s_{ij_t}\}$ is a set of option-value pairs $s_{ij} =< o_i, v_j >$ for all $1 \leq t \leq n$, where all options are distinct, $o_i \in O$ for $i = 1, 2, \ldots, n$ and $v_j \in V_i$, $j = 1, 2, \ldots, n$*

For given any system model $M =< O, V, Q >$, the requirements list can be constructed by enumerating every possible t-length option-value pair. However, in some cases, there may be some specific option-value pairs which may not be allowed in any configuration. These t-tuples are called invalid t-tuples or constraints $q_i \in Q$. Every t-tuple which contains or is equal to any constraint, has to be excluded from requirements list.

Definition 2 *A configuration is a n-tuple option-value pairs $s_{ij} =< o_i, v_j >$, where every option is included in one of the option-value pairs exactly once. A valid configuration is a configuration where none of the invalid t-tuples are included in the configuration.*

As we described in Section 2.1, after SUT is modeled, in the second phase of CIT all t-tuples are sampled into small number of groups such that from each group a valid configuration can be constructed. These sampled requirements form a test suite for the SUT and t is called the strength of the test suite.

Definition 3 *A t-way Covering Array is $CA(N; k, t, M =< O, V, Q >)$ an N×k array with N valid configurations and k columns (number of options) where each valid t-tuple requirement $R_i \in R$ is covered at least once by any configuration.*

An example of 2-way Covering Array with 5 options each having binary values is given in Figure 2.2. Every possible 2-tuple of option combinations is present in the array.

o_0	o_1	o_2	o_3	o_4
0	1	0	0	0
1	0	0	1	0
1	1	0	1	1
0	0	1	1	1
1	1	1	0	1
1	0	1	0	0

Figure 2.2: A binary 2-way Covering Array with 5 options.

In this thesis, we propose several methods to construct t-way CAs in the presence of constraints in a smaller size and a reasonable time.

2.3. Simulated Annealing

Simulated Annealing (SA) is a generic probabilistic method for the global optimization first introduced by Kirkpatrick et al. [35] and Cerny [9]. Originally, SA concept was influenced by a correlation between the physical annealing process of solids as a thermal process and the problem of solving large combinatorial optimization problems.

The annealing process consists of two steps [35]. In the first step, the temperature of heat bath is increased to a maximum value T_0 at which the solid melts and in the second phase the temperature of the heat bath is decreased with a cooling rate C_r until the molten metal gets frozen or reached to a desired temperature T_s.

In annealing process, it is very important to keep the potential energy state at minimum of solid in a heat bath. Because, at high temperatures particles arrange themselves randomly, on the other hand, particles are arranged to be highly structured when the corresponding energy is minimal. Hence, as the SA process continues, particles get stabilized more and it becomes difficult to make big structural changes. The annealing process is terminated

if the temperature reaches to stopping temperature T_s or potential energy becomes 0. In annealing, it is very important to chose C_r carefully because if C_r is not small enough frozen metal will contain imperfections caused by unreleased energy and if the cooling rate is small then the frozen metal will be too softened to work with.

SA uses the same idea of annealing thermal process to solve the global optimization problems. Energy and the state of minimum potential energy in annealing process correspond to the gain and optimal solution with maximum gain in SA, respectively. The annealing parameters T_0, C_r, and T_s are chosen with respect to problem domain to control the search process. The local minima optimization heuristic search methods choose the best available option in the current state to find the optimal solution. These techniques may be quite efficient while global optimal solution is not needed. However, SA differs from other local minima heuristic search methods in accepting or declining the state using a probabilistic method called Boltzmann Distribution function (2.1).

$$B(T) = -k_b \frac{\Delta E}{T} \tag{2.1}$$

In the decision step, if the gain is not negative, it is accepted in any case. However, if it is smaller than 0, the decision is carried out by the truth value of (2.2). The right side of the inequality decreases as the T value goes down so it becomes difficult to accept a state as the state cools down more.

$$Rand(0, 1) < e^{B(T)} \tag{2.2}$$

The main reason of why we choose SA in our proposed approach is that CA is a combinatorial object and combinatorial problems are indeed counting problems. We conjecture that counting is a simple task, so that it can be done in parallel to decrease the execution time. Moreover, we believe that the SA algorithm is suitable to be implemented with parallel computing techniques. That's why we adapt SA to our work. The complete algorithm is explained and discussed more extensively in Section 4.5.

2.4. CUDA

CUDA (Compute Unified Device Architecture) is a parallel programming model created by NVIDIA. CUDA allows programmers to increase the computing performance with thousands of CUDA-enabled graphics processing units (GPUs). GPUs can be used via CUDA-accelerated libraries, compiler directives (such as OpenACC), and extensions to industry-standard programming languages, including C, C++ and Fortran. In this thesis we use C/C++ programming language with NVIDIA's LLVM-based C/C++ compiler (nvcc).

In CUDA programming model, programs are structured in such a way that some functions are executed on the CPU which is called host, while some functions are executed on the GPU which is referred as the device in the context of CUDA. The code to be executed by the CPU, schedules kernels (GPU functions) to be executed on the device. Therefore, CUDA programming paradigm is a combination of sequential and parallel executions, and is called heterogeneous type of programming.

CUDA manages parallel computations using the abstractions of threads, blocks and grids. A *thread* is just an execution of a kernel with its unique index. A *block* is a set of threads. Threads within the same block can be synchronized using $\_syncthreads()$ which makes threads wait at a certain point in the kernel until all the other threads within the same block reach the same point. A *grid* is a group of blocks where no synchronization exists at all between the blocks in device level.

The heterogeneous architecture of CUDA is given in Figure 2.3. Sequential code invokes the kernel function from CPU with specifying number of threads in a thread block and number of blocks in a grid. Grid and block variables are written in three angular brackets $<<< grid, block >>>$ before providing inputs to the kernel as shown in Figure 2.3. In this invocation, grid and thread blocks are created and scheduled dynamically in the hardware level. The value of this grid and block variables must be less than the allowed sizes.

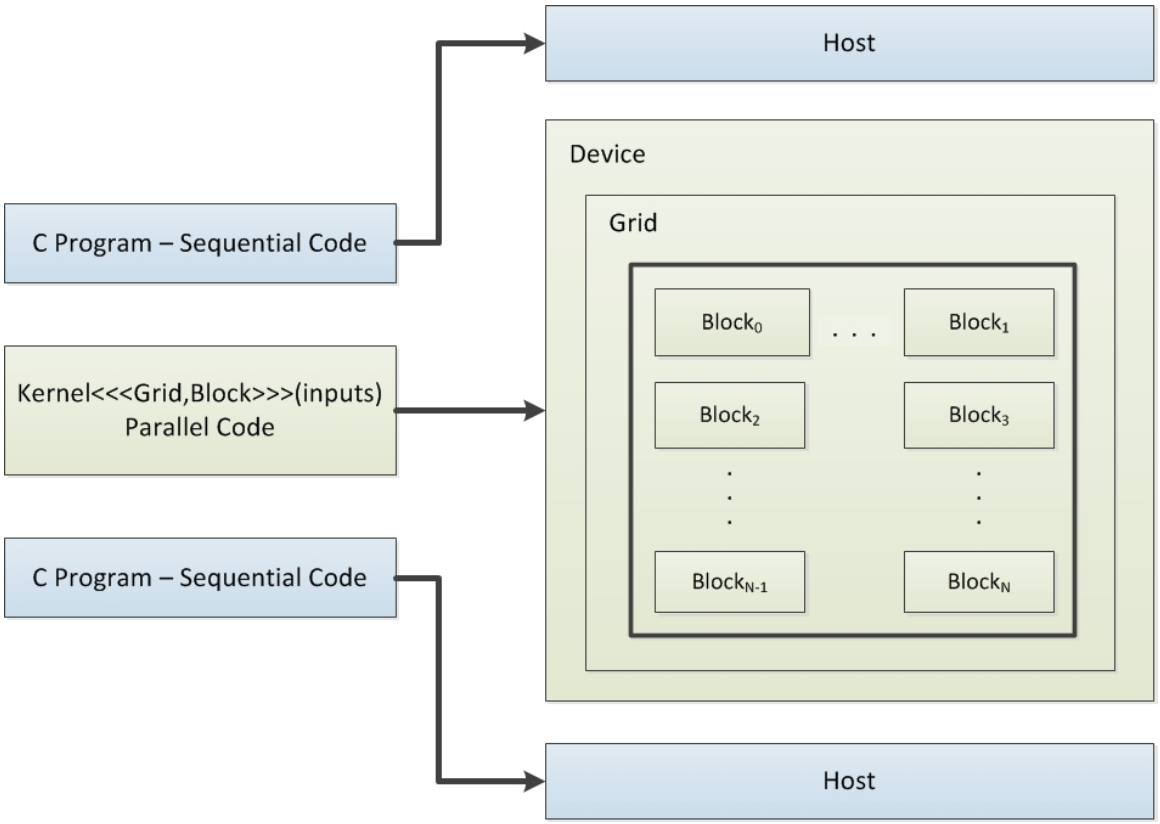

Figure 2.3: Heterogeneous structure

Once a block is initialized, it is divided into groups having 32 threads. These units of 32 threads form *warps*. All threads within a same warp must execute the same instruction at the same time, i.e., instructions are handled per warp. This issue arises a problem called *branch divergence*. It happens when threads inside warps branches to different execution paths and this forces the paths to be executed sequentially. In other words, every thread in a warp has to execute the same line of code. Hence, it is important to assign the same jobs to the threads within the same warp.

A kernel can be launched using thousands or even millions of lightweight threads that are to be run on the device. CUDA threads are thought of as lightweight because of that they have almost no creation overhead, meaning that thousands can be created quickly. The scheduling of the thread execution and thread blocks is also handled on the hardware.

There are several types of memories on GPUs; global memory, shared memory and registers. Global memory is used to copy data from CPU to GPU or GPU to CPU. It has the largest memory among the others, however the slowest on reading and writing data. On the other hand, shared memory can be thought as a cache memory of the blocks in GPU. Every block has its own shared memory and it is not accessible by other threads in other blocks. Shared memory can be read and written only from the device part. Registers are sometimes referred as local memories of threads. They are the fastest ones on data reading and writing but have less memory, too.

We give an example of a simple code to understand the need for CUDA. Consider a normal sequential C program performing vector addition given below. Every addition is done sequentially.

```
float vectorA [3] = {1.3, -1.3, -1.0};
float vectorB [3] = {1.4, 3.5, 11.2};
float vectorC [3];
for(int i; i < 3; i++)
        vectorC[i] = vectorA[i] + vectorB[i];
```

On the other hand, in efficient CUDA programs, data is arranged well organized so that each thread can share the work to be done. An example of CUDA code that does the same job as the above sequential C program is given below.

```
float vectorA [3] = {1.3, -1.3, -1.0};
float vectorB [3] = {1.4, 3.5, 11.2};
float vectorC [3];
int i = threadIdx.x;  # threadIdx: thread index
vectorC[i] = vectorA[i] + vectorB[i];
```

Each position in *vectorA* and *vectorB* is added to *vectorC* in parallel by different threads, i.e., the same code is executed by each tread but different positions.

More detailed information about CUDA architecture and memory management are given in [32].

2.5. Boolean Satisfiability Problem

Boolean satisfiability testing (SAT) is a problem to decide whether a given Boolean formula has any satisfying truth assignment. SAT is the first problem that is proven as NP-complete by Cook [17]. Nowadays, there are many efficient SAT solvers. They try to replace the variables of the given Boolean formula with *true* or $false$ values in a way that the formula is evaluated as *true*. If any such values can be found which makes formula *true*, then the formula is considered as $satisfiable$ i.e., there exist at least one solution for the problem. On the other hand, if no values can be found in order to make formula *true*, then it is called as $unsatisfiable$. As an example, the formula "$\neg x_1 \vee x_2$" is satisfiable because when $x_1 = false$ and $x_2 = false$, "$\neg x_1 \vee x_2$" becomes *true*. However, no solution exists for the formula "$\neg x_1 \wedge x_1$", since every assignment makes the formula $false$. Hence, the formula "$\neg x_1 \wedge x_1$" is called $unsatisfiable$

In SAT solvers, formulas are represented in conjunctive normal form (CNF) which is conjunction of clauses. A *clause* is a disjunction of literals and a *literal* is either a variable, or the negation of a variable. Basically, the main goal of SAT is to find values for all variables which makes each of these clauses *true*.

SAT solvers have been also studied commonly to construct CAs [3, 12, 13, 30, 40, 56]. However, their scalability issue is a hard problem that makes these approaches mostly impractical. In our study, to avoid scalability issue and benefit from SAT solver, we use it to generate a valid configuration which covers only the provided t-tuples. This functionality is used to add the missing t-tuples of CA in order to make it complete. More detailed explanation is given in Section 4.7.

3

RELATED WORK

Nie et al. [41] points out that CA construction is an NP-hard problem, so it attracts many researchers attention from various fields. Much research has been done to develop efficient techniques and tools for constructing CAs with small size in a reasonable time. They collect more than 50 work on CA construction and classify those proposed techniques into 4 groups: greedy algorithms [5, 7, 10, 19, 38, 49, 52], heuristic search algorithms [8, 11, 15, 24, 46], mathematical methods [29, 36, 53, 54], and random search-based methods [27, 45].

Greedy algorithms [5, 7, 10, 19, 38, 49, 52], as the name suggests, perform in a greedy way, i.e., they construct CAs iteratively by choosing the best case scenario to cover more uncovered t-tuples among all possible choices in each iteration. In general, these type of algorithms choose the best available configuration or generate a new configuration which covers most of the uncovered t-tuples until no t-tuple is left uncovered. Greedy algorithms have been the most widely used approach for test suite generation in CIT.

Moreover, heuristic search techniques such as hill climbing [15], great flood [8], tabu search [8,25], particle swarm optimization [55] and simulated annealing [11,51] as we did in our research, have been used in many work. In addition to these techniques, some AI-based approaches have been also employed to constrcut CAs, e.g., genetic algorithm [24] and ant colony algorithm [46]. In general, heuristic search methods start from a non-

complete CA state and apply some operations on the state iteratively until no t-tuple left uncovered or reach to a threshold. One of the main advantages of these techniques is that they do not require searching the whole space. Nonetheless, they are poor in finding optimal values, but, they show great efficiency both in time and size in many work.

Random based search method is also used in CA construction [27, 45]. These techniques randomly select configurations from a complete larger set until all t-tuples are covered. In some of special cases, random search may produce better results than other methods.

Besides these techniques, several mathematical approaches [29, 36, 53, 54] are also proposed for CA construction. These techniques have been studied by researchers mainly from mathematical fields. These approaches are mostly extended versions of methods for constructing orthogonal arrays [43].

In practical scenarios, many SUTs have system-wide inter-option constraints and existence of these constraints makes even harder to construct smaller size CAs in a reasonable time. Therefore, constraint handling is another problem which is extensively studied in CIT. Bryce et al. [6] presented an approach to handle with "soft constraints" and then, Hinc et al. [30] proposed a technique for "hard constraints", even though they only provided small scale of inputs and their approach was not scalable. Cohen et al. [13, 14] introduced new techniques to deal with constraints. They described several types of constraints which may be present in highly configurable systems. They presented an approach to encode the constraints into SAT problem. Mats et al. [26] provided four techniques to handle constraints as well as giving the weaknesses of techniques in order to choose the best one when needed. In our study, we consider constraints as invalid t-tuples and try to construct CAs excluding these t-tuples.

Recently, several parallel computing approaches have been also proposed to construct CAs. One example of that is studied by Avila [1]. The author presents a new improved SA algorithm for CA construction and various ways to employ multiple SA in parallel using grid computing. The author does not really parallelize the SA algorithm, but gives several approaches to run SA algorithms in parallel. Another work is done by Younis et al. [60]. They present a new parallel strategy based on earlier method IPOG, called multicore modified input parameter order (MC-MIPOG). Unlike IPOG, MC-MIPOG apply a novel

approach by eliminating control and data dependency to let the utilizing of multicore systems. Lopez [39] presents a parallel algorithm for the software product line testing. The author uses a parallel genetic algorithm in order to construct CAs and evaluates the algorithm with comparing similar approaches. In our approach, we propose our algorithm for GPU-based parallel computing techniques to make it easier for any user who has a computer with GPU. Moreover, since constructing a CA is a simple counting problem, we believe that rather than having less number of cores with high capability, having more cores with less capability can be more effective. That's why we propose our algorithms for GPUs.

Due to the fact that CIT is getting used more widely in practical cases, several tools are developed to construct CAs effectively [18]. We investigate and make comparisons with 4 well known tools: ACTS, CASA, PICT, Jenny. Advanced Combinatorial Testing System (ACTS) is developed jointly by the US National Institute Standards and Technology (NIST) and the University of Texas at Arlington. It can generate CAs with strengths 2-way through 6-way and also supports for constraints and variable-strength tests as well. CASA is developed by Garvin et al. [22, 23]. They use the same heuristic search method SA as we did in our work but with sequential algorithms. CASA can deal with logical constraints explicitly. Another tool to construct CA is PICT developed by Microsoft. They claim that PICT was designed with three principles: (1) speed of CA generation, (2) ease of use, and (3) extensibility of the core engine. The ability to generate the smallest size CA is given less emphasis. Jenny [33] is another well known tool in this area. It also supports constraint handling and constructing variable strength CAs. Our proposed approach supports constraint handling too, but not variable strength. Nonetheless, due to the nature of heuristic methods, they suffer in time as the space get larger. We try to overcome this problem using parallel computing. Moreover, we combine heuristic searh and SAT solver to improve the quality further, especially for larger configuration spaces. We show that our proposed algorithm can construct smaller size CAs in a reasonable time.

4

METHOD

This chapter discusses the details of the proposed approach to measure the combinatorial coverage of any given array and steps to construct CAs in the presence of system constraints within option interactions both with sequential and parallel algorithms. Moreover, we give an approach to generate multiple neighbour states in parallel and a hybrid approach to increase the time efficiency and size quality.

In Section 4.1, an overview of simulated annealing for construction of CAs is described. Section 4.2 explains the outer search algorithm. An initial strategy is presented in Section 4.3. In the following sections, sequential and parallel algorithms are explained for Combinatorial Coverage Measurement (Section 4.4) and Simulated Annealing (Section 4.5). Moreover, we also present a method to generate multiple neighbour states in parallel (Section 4.6) and propose a novel hybrid approach to construct CAs using a SAT solver (Section 4.7).

4.1. Method Overview

In our SA definition, a state refers to a set of N valid configurations where N is the size (number of rows) of the state. NS is a next state of current state whose one option value in one configuration is changed to another value. Furthermore, the fitness function (gain function) counts the difference between number of uncovered t-tuples for the current state and the NS.

SA can not modify the size of the state while trying to cool down the current state from initial temperature to final temperature. In other words, SA can neither add extra configurations nor remove any of them from the state. This cooling down process is sometimes called as inner search, as well.

Besides that, deciding a size for an inner search is also a difficult problem. However, finding tight lower and upper bounds is not an easy task. Many work has been done to determine good upper and lower bounds for the CA size $[16, 21, 31, 42, 48, 61]$ in order to converge to the optimal size faster. However, especially when the option values are not binary, the gap between bound estimations is not good enough to approximate the optimal size. Therefore, we need an outer search algorithm which calls the inner search algorithm repeatedly while choosing the next state size more systematically. This next state size decision process has to decrease the gap between the bounds in each iteration as much as possible in order to avoid calling inner search many times.

In the following sections, we explain the outer search and inner search algorithms in detail.

4.2. Outer Search

As we described above, in this section, we present an outer search algorithm to construct CAs as given in Algorithm 1.

Algorithm 1 Covering Array Generation 1

Input: $M =< O, V, Q >$: SUT Model, t: strength, P_0: initial temperature, P_f: final temperature

Output: $CA(N; k, t, M =< O, V, Q >)$: t-way Covering Array

```
1:  B_l ← 0, B_u ← INT_MAX
2:  isLowBoundFound ← false
3:  isUpBoundFound ← false
4:  N ← estimateSizeOfCA(t, M)                    # Number of rows (confgurations)
5:  S_0 ← generateInitialState(N, M)
6:  S_l ← NULL, S_u ← NULL                        # lower and upper boundary states
7:  N_u ← combinatorialCoverageMeasurement(S_0, t, M)
8:  S ← S_0
9:  while (true) do
10:     S, N_u ← simulatedAnnealing(S, N, N_u, t, M, P_0, P_f)
11:     if (N_u > 0) then
12:         B_l ← N
13:         isUpBoundFound ← true
14:         S_l ← updateBoundaryState(B_l, S)
15:     else if (N_u = 0) then
16:         B_u ← N
17:         isLowBoundFound ← true
18:         S_u ← updateBoundaryState(B_u, S)
19:     end if
20:     if (B_u − B_l < 2) then                   # Minimal size is found, B_u
21:         break
22:     end if
23:     N ← nextStateSize(N_u, B_u, B_l, isLowBoundFound, isUpBoundFound)
24:     S ← updateCurrentState(S_l, N, N_u, isLowBoundFound)
25: end while
26: return S_u
```

In Algorithm 1, we provide the specifications of SUT and strength t as inputs to the algorithm. As the first step, the algorithm marks the lower and upper bounds as not found (line 2-3). Then, since the inner search needs N configurations to start the search algorithm, an initial state (IS) has to be provided to the system. Algorithm determines a size for the IS using ACTS. Based on our experiments, we observed that ACTS constructs CAs very fast when system has no constraint. Hence, first, we run ACTS experiment on the same M but without constraint version, and then, assign 80% of ACTS result to initial size of CA. After that, M and N are provided to the IS generation phase (lines 4-5). IS generation phase uses the hamming distance (HD) approach to generate an initial state (Section 4.3). HD aims to keep the number of uncovered t-tuples of the IS as small as

possible while constructing the state. In the next step, using Combinatorial Coverage Measurement (CCM) algorithm (Section 4.4), the number of missing (uncovered by any configuration) t-tuples of the IS is counted (line 7). For the inner search algorithm, SA attempts to construct a t-way CA with the given inputs (Section 4.5). If SA succeeds to construct a complete t-way CA, then this size is marked as upper bound and upper boundary state S_u is updated otherwise, i.e., cannot cover all t-tuples, the size is marked as a lower bound and lower boundary state S_l is updated (line 10-19). Outer search loop (line 9-25) continues until the minimal CA size is found, i.e., the difference between upper and lower bounds is smaller than 2.

Algorithm 2 is used to determine the next state size of CA. If an upper or lower bound is not found yet, we simply return 90% or 110% of boundary size, respectively. If both of the bounds are found, we estimate 4 different sizes and choose the minimum one. N_1 mimics simple binary search technique and N_4 assumes that each configuration covers $(N_u/((k/t) + 1))$ t-tuples at most.

The following sections give detailed explanations about the functions which are used in the proposed approach and present both sequential and parallel algorithms for them.

Algorithm 2 Next State Size for CA

Input: N_u: number of uncovered t-tuples, B_u: upper bound, B_l: lower bound, $isLowBoundFound$, $isUpBoundFound$
Output: N : Size of the next state CA

1: **if** (!$isLowBoundFound$ *and* $isUpBoundFound$) **then**
2: $N \leftarrow B_u \times 0.90$
3: **else if** ($isLowBoundFound$ *and* !$isUpBoundFound$) **then**
4: $N \leftarrow B_l \times 1.10$
5: **else if** ($isLowBoundFound$ *and* $isUpBoundFound$) **then**
6: $N_1 \leftarrow (B_u + B_l)/2$
7: $N_2 \leftarrow B_l \times 1.10$
8: $N_3 \leftarrow B_u \times 0.90$
9: $N_4 \leftarrow (N_u/((k/t) + 1)) + 2$
10: $N \leftarrow min(N_1, N_2, N_3, N_4)$
11: **end if**
12: **return** N

4.3. Initial State Generation

The main idea in the SA algorithm is to decrease the number of missing t-tuples until no t-tuple is left uncovered. Therefore, starting with a better IS which covers more t-tuples will probably decrease the search time to construct a complete CA if it exists with the given size.

In the literature, HD is widely used for this purpose [47, 51]. We also use HD in our proposed work but with a different technique based on an observation: in complete CAs, every setting values in each option column is distributed to along the column equally sized as much as possible. Actually, HD performs in a similar way, i.e., it keeps the similarity between each configuration large and leads having an equally size settings distribution along the columns. Therefore, in order to construct a similar structure initially, we developed 2 strategies for different cases of SUT e.g., with constraints and without constraints.

While the system has no constraint, we generate an IS in such a way that every option values are distributed randomly to the corresponding column almost equally sized. The procedure is given as follows:

For each option column;

1. Find the number of possible values
2. Generate columns consisting of equally sized option values
3. Randomize the order of values in the columns

In the presence of constraints, it not possible to use the same idea since not every configuration becomes valid. Therefore, we use a similar modified approach. The procedure is given as follows:

1. Generate 2 times more valid configurations than needed
2. Pick the first configuration
3. Iterate over other configurations and pick the one which makes the option columns settings distribution equally sized

Since this phase is done only once and the given sequential strategies perform well enough, we did not parallelize this phase.

4.4. Combinatorial Coverage Measurement

CCM is employed in the outer search algorithm to count the number of uncovered t-tuples of IS to proceed with the inner search. It can also be used while checking any CA is whether complete or not. If CA is not complete, it returns the coverage percentage of the given CA. Moreover, it may also be required in some cases during the computation not just before the computation. Because of these facts, faster CCM calculation carries big importance in CA construction.

Before describing the algorithms for CCM, we try to explain how CCM calculation is done indeed.

In CCM, every possible t-way option combination is investigated to measure the combinatorial coverage of CA i.e., the number of covered t-tuples of each option combination has to be counted. For example, consider the 2-way binary CA given in Figure 4.1.(b) for the configuration space model Figure 4.1.(a) without any constraints between any option interactions. There exists 6 possible 2-way option combinations $\{< o_1, o_2 >, < o_1, o_3 >, < o_1, o_4 >, < o_2, o_3 >, < o_2, o_4 >, < o_3, o_4 >\}$ for this configuration space model and since every option takes 2 values {0, 1}, for each option combination, there are 4 possible 2-tuples as shown in Figure 4.2. These all 2-tuples have to be covered in each option combination for CA to be complete. Therefore, in general, in order to measure the combinatorial coverage, i.e., count the number of covered t-tuples (or uncovered t-tuples), every possible t-tuple of every option combination is checked whether it is present in the CA or not.

In the following subsections, we give both sequential and parallel algorithms which are inspired from [2] to find the number of uncovered t-tuples of the CA.

Configuration Space Model	
option	**settings**
o_1	{0, 1}
o_2	{0, 1}
o_3	{0, 1}
o_4	{0, 1}

(a)

2-way CA			
o_1	o_2	o_3	o_4
1	1	0	0
1	0	1	1
0	1	1	0
0	0	0	1
0	1	0	1
1	0	0	0

(b)

Figure 4.1: A configuration space model (a) and a covering array (b) for this model.

2-tuple	
o_i	o_j
0	1
1	0
1	1
0	0

Figure 4.2: All possible binary 2-tuples for the option combination of o_i and o_j

4.4.1. Sequential combinatorial coverage measurement

The sequential approach given in Algorithm 3 is described as follows. As initial, the number of t-way option combinations is found by calculating the number of all different ways to choose t options out of k (4.1). Then, the number of all valid t-tuples is counted by excluding the invalid t-tuples from all t-tuples and it is assigned as number of uncovered t-tuples N_u (line 2). Afterwards, the number of maximum settings among all option's settings is marked (line 3).

$$C(k,t) = \frac{k!}{t!(k-t)!} \; where \; t \leqslant k \; and \; t > 0 \tag{4.1}$$

Iterating over option combination indices, every t-length option combination $< o_{i_1}, o_{i_2}, \ldots, o_{i_t} >$ is generated one by one (line 5) and number of covered t-tuples for each option combination is assigned to 0 (line 6). Then, all entries of the lookup table is initialized to $false$ indicating that no t-tuple is covered for the corresponding option combination, yet.

Algorithm 3 Sequential Combinatorial Coverage Measurement

Input: S: CA state, $M =< O, V, Q >$: System Model, N: size of CA, t: strength
Output: N_u: number of uncovered t-tuples

1: $N_{optComb} \leftarrow numberOfCombinations(k, t)$ # choose t out of k
2: $N_u \leftarrow findNumberOfValidTuples(t, M)$
3: $N_{max|V_i|} \leftarrow max_{i \in [0,k-1]}|V_i|$
4: **for** $(i \leftarrow [0, \ldots, N_{optComb} - 1])$ **do**
5: $\quad optComb_i \leftarrow generateOptCombFromIndex(k, t, i)$
6: $\quad N_u' \leftarrow 0$
7: $\quad lookupTable \leftarrow initializeLookupTable(N_{max|V_i|})$
8: $\quad$ **for** $(r \leftarrow [0, \ldots, N - 1])$ **do** # r: row id
9: $\quad\quad R_r \leftarrow S[r][\forall o_j \in optComb_i]$
10: $\quad\quad H_r \leftarrow convertTupleToNumber(R_r)$
11: $\quad\quad$ **if** $(!lookupTable[H_r])$ **then**
12: $\quad\quad\quad lookupTable[H_r] \leftarrow true$
13: $\quad\quad\quad N_u' \leftarrow N_u' + 1$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: $\quad N_u \leftarrow N_u - N_u'$
17: **end for**
18: **return** N_u

The lookup table keeps track of which t-tuples are covered in the array. Size of lookup table is calculated as follows;

$$sizeof(lookupTable) = \prod_{j=1}^{t} N_{max|V_i|} \tag{4.2}$$

In this way, we ensure that the size of lookup table is greater or equal to number of all valid t-tuples for each option combination.

After an option combination is generated and the lookup table is initialized, the corresponding columns to the generated option combination $< o_{i_1}, o_{i_2}, \ldots, o_{i_t} >$ are picked from CA and an $N \times t$ table is constructed, virtually. Every t-tuple (lines in $N \times t$) is scanned in this table and is mapped into a unique index using a hash function (4.3) (line 10).

$$H(R_i) = \sum_{i=1}^{t} v_j \times (N_{max|V_i|})^{i-1} \tag{4.3}$$

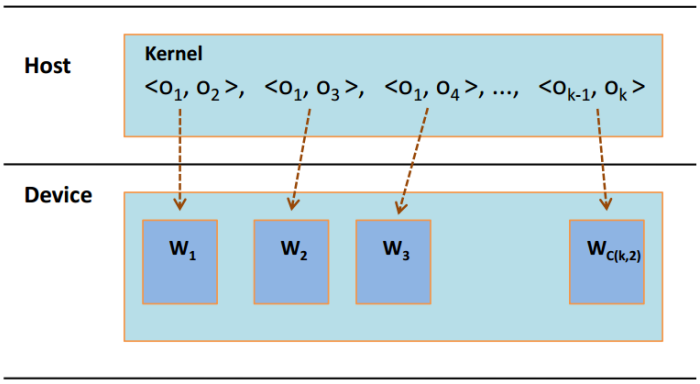

Figure 4.3: 2-way option combination distribution between warps

This hash function basically considers the t-tuple R_i in the base of $N_{max|V_i|}$, and converts it to the base of 10. Since $N_{max|V_i|}$ is greater or equal to all options' settings, uniqueness of this conversion is guaranteed.

Once the hash index of t-tuple is computed, this index entry in the lookup table is checked. If it is assigned to *true*, then it is already covered by another configuration. On the other hand, if the corresponding t-tuple is assigned to $false$, then this t-tuple is not covered by any configuration before. Therefore, the number of covered t-tuples is increased by 1 and this index entry in the lookup table is assigned to *true* (lines 11-14). After scanning each line of $N \times t$ table, the number of covered t-tuples for the corresponding option combination is subtracted from the number of uncovered t-tuples (line 16). This procedure is repeated for every option combination. Finally, the algorithm returns the number of uncovered t-tuples N_u for the given CA.

4.4.2. Parallel combinatorial coverage measurement

In the parallel approach of the CCM, we use the same idea with the sequential algorithm only with a difference. We observe that counting missing t-tuples of any option combination is independent of another, i.e., measuring the coverage of any option combination does not affect the other option combinations coverage. Hence, investigation of uncovered t-tuples for every option combination can be done separately.

We use this idea to parallelize the approach to improve the time efficiency. To do so, we distribute every option combination to different warps with respect to their warp indices. In the above example (Figure 4.1), option combinations $\{< o_1, o_2 >, < o_1, o_3 >, < o_1, o_4 >, < o_2, o_3 >, < o_2, o_4 >, < o_3, o_4 >\}$ are sent to $w_0,\ w_1,\ w_2,\ w_3,\ w_4$ and w_5, respectively as shown in Figure 4.3.

In the first step of the approach (Algorithm 4), we collect all needed information such as maximum number of blocks B_{max} and threads T_{max} of the available GPU device in order to get full performance (lines 1-2). T_{max} is 1024 for almost all new GPU devices and B_{max} changes with respect to device capability. In our case, T_{max} is also 1024 and B_{max} is 32. So, we have 32 warps in each block and 1024 warps (N_w) in the entire grid. Using N_w information, every warp generates its option combinations as in (4.4) and counts the missing t-tuples of these option combinations iteratively.

$$optCombs_{w_i} = \{optComb_j \mid j \equiv i\ mod(N_w),\ where\ 0 \leq j < N_{optComb}\} \tag{4.4}$$

Before the kernel is launched, we specify the size of shared variables for a single block. Since 32 option combinations are investigated in a single block at the same time, size of lookup table is increased by 32 (4.5).

$$sizeof(lookupTable) = 32 \times \prod_{j=1}^{t} N_{max|V_i|} \tag{4.5}$$

Algorithm 4 Parallel Coverage Measurement of CA

Input: S: CA state, $M =< O, V, Q >$: System Model, N: size of CA, t: strength
Output: N_u: number of uncovered t-tuples

1: $B_{max} \leftarrow getMaxNumberOfBlock()$
2: $T_{max} \leftarrow getMaxNumberOfThreadEachBlock()$
3: $N_u \leftarrow 0$
4: $N_{optComb} \leftarrow numCombinations(k, t)$
5: $N_{max|V_i|} \leftarrow max_{i \in [0,k-1]}|V_i|$
6: $sizeLookup \leftarrow {N_{max|V_i|}}^t$
7: $CCMKernel <<< B, T >>> (CA, t, M, N_u, sizeLookup)$
8: $cudaDeviceSynchronize()$
9: **return** N_u

The *CCMKernel* function is given in Algorithm 5. This algorithm is executed for every block and every thread.

In Algorithm 5, as the first step, warp id across all blocks ($wGrid_{Id}$) and warp id within the block ($wBlock_{Id}$) are calculated for all warps. Also, the thread id within the warp ($Twarp_{Id}$) is calculated for all threads. $wGrid_{Id}$ is used for option combination distribution: every warp knows which option combinations they are responsible for, by their $wGrid_{Id}$. The parameter $wBlock_{Id}$ is used to change or check the lookup table. Since lookup table is defined as shared variable, i.e., only visible and modifiable by the threads within the same block, an indexing method is needed for warps within the same block.

We initialize a local variable named ${N_u}^{T_{Id}}$ to 0 for every thread (line 5). This variable counts the number of uncovered t-tuples only for the corresponding thread. Finally, all ${N_u}^{T_{Id}}$ variables are added to N_u. Keeping this ${N_u}^{T_{Id}}$ variable as local for each thread helps us to avoid writing it to global variable N_u in each iteration.

At the beginning, $comb_{Id}$ is initialized to $wGrid_{Id}$. Then, counting process for the option combination whose index is $comb_{Id}$, is started for every warp. After counting uncovered t-tuples of the first option combination, $comb_{Id}$ is incremented by N_w in each iteration as in (4.4) (line 31).

In the counting process of any option combination, the first thread in the warp generates the corresponding option combination (lines 9-11) and then, threads within the same warp initialize the part of the lookup table which is specified for them, to $false$ (lines 13-16). An $N \times t$ table is constructed with columns of options in option combination, virtually same as with sequential algorithm. Each line (t-tuple) of the $N \times t$ table is checked by the thread with the corresponding $Twarp_{Id}$ in parallel. Every thread converts the corresponding t-tuple into a hash index using (4.3) and assign the hash index position in the lookup table as *true*. If the number of rows are greater than 32, then each thread checks more than 1 row until every row is scanned (lines 18-23).

Algorithm 5 CCM Kernel

Input S: CA state, $M =< O, V, Q >$: System Model, N: size of CA, t: strength
N_u: number of uncovered t-tuples, $sizeLookup$: size of lookup table
Output N_u: number of uncovered t-tuples

1: $wGrid_{Id} \leftarrow T_{Id}/32$ # $wGrid_{Id}$: warp id in grid
2: $wBlock_{Id} \leftarrow (T_{Id}\%1024)/32$ # $wBlock_{Id}$: warp id in block
3: $Twarp_{Id} \leftarrow T_{Id}\%32$ # $Twarp_{Id}$: thread id in warp
4: $comb_{Id} \leftarrow wGrid_{Id}$
5: ${N_u}^{T_{Id}} \leftarrow 0$
6: $N_{optComb} \leftarrow numberOfCombinations(k, t)$ # choose t out of k
7: **shared bool** $lookupTable[N_w * sizeLookup]$
8: **while** $(comb_{Id} < N_{optComb})$ **do**
9: **if** $(Twarp_{Id} = 0)$ **then**
10: $optComb_{w_{Id}} \leftarrow generateOptCombFromIndex(comb_{Id})$
11: **end if**
12: $i \leftarrow Twarp_{Id}$
13: **while** $(i < sizeLookup)$ **do**
14: $lookupTable[wBlock_{Id} \times sizeLookup + i] \leftarrow false$
15: $i \leftarrow i + 32$
16: **end while**
17: $r \leftarrow Twarp_{Id}$
18: **while** $(r < N)$ **do**
19: $R_r \leftarrow S[r][\forall o_j \in optComb_{w_{Id}}]$
20: $H_r \leftarrow convertTupleToNumber(R_r)$
21: $lookupTable[wBlock_{Id} \times sizeLookup + H_r] \leftarrow true$
22: $r \leftarrow r + 32$
23: **end while**
24: $i \leftarrow Twarp_{Id}$
25: **while** $(i < sizeLookup)$ **do**
26: **if** $(lookupTable[wBlock_{Id} \times sizeLookup + i] = true)$ **then**
27: ${N_u}^{T_{Id}} \leftarrow {N_u}^{T_{Id}} + 1$
28: **end if**
29: $i \leftarrow i + 32$
30: **end while**
31: $comb_{Id} \leftarrow comb_{Id} + N_w$
32: **end while**
33: $N_u \leftarrow N_u + {N_u}^{T_{Id}}$

After scanning all t-tuples of the option combination, every thread checks different position in the lookup table to identify the missing t-tuples. If the position is false, i.e., the corresponding t-tuple is not covered by any option combination, so ${N_u}^{T_{Id}}$ variable increased by 1 (lines 25-30). Once every option combination is processed, each thread adds its ${N_u}^{T_{Id}}$ variable to N_u using atomic operations of CUDA. The algorithm finally returns the variable N_u.

4.5. Simulated Annealing For Constructing Covering Arrays

We use SA to construct a CA with the given size by applying several random changes on a given state. It attempts to decrease the number of uncovered t-tuples of the given state without counting them in each iteration. Therefore, CCM algorithm is called only once before we begin the outer search to measure the combinatorial coverage of the given state.

SA takes SUT specifications, number of configurations, current state, number of uncovered t-tuples as inputs and aims to construct a complete CA. If the algorithm cannot construct a complete CA, it returns the state whose temperature is the final temperature. Otherwise, the algorithm returns a complete CA.

As the temperature values of SA process, 1 and 0.0001 are assigned to P_0 (initial temperature) and P_f (final temperature), respectively. On the other hand, due to the fact that our scale of experiments varies from strength 2 with 20 options to strength 3 with 200 options choosing a fixed value for the cooling rate R does not serve our propose well. Therefore, we come up with a new R formula depending on the configuration space variables as follows:

$$R = \frac{1 - 0.001}{0.15 \times (10^t \times k \times t)} \tag{4.6}$$

Since determining an optimal value for R is beyond the scope of this work, we just made a set of small-scale experiments to come up with this formula. We believe that based on our experiments, the number of inner loop needs to close to the number $(10^t \times k \times t)$. In order to approximate to this number, we develop the formula given in (4.6) for R.

There are 2 main phases in the SA algorithm, called neighbour state (NS) generation and fitness function. In NS generation phase, we generate randomly NSs choosing a random position in CA and a random new value for this position. Then, using fitness function, we count the change (gain) in the number of uncovered t-tuples between the current state and NS. Then, the gain is provided to a decision function to accept or reject the NS. We describe NS generation phase and fitness function in detail with both sequential and parallel algorithms in Section 4.5.2 and Section 4.5.3.

In the decision step (Algorithm 6), if the gain is not negative i.e., the number of uncovered t-tuples of NS is lower than or equal to the current state's, NS is accepted in any case. However, if accepting NS increases the number of uncovered t-tuples, we use a probabilistic function given in (4.7) to decide whether to accept the NS or not. Since decision function also cares about the current temperature, as the temperature of SA cools down, the *probValue* decreases and accepting a costly state becomes difficult. This step saves SA to get trapped in local minima or maxima.

$$B(C, P) = -k_b \frac{C}{P} \tag{4.7}$$

Algorithm 6 Neighbour State Decision

Input: C : gain, P : current temperature
Output: *true* or $false$

1: **if** $(C > 0)$ **then**
2: **return** *true*
3: **end if**
4: $randNumber \leftarrow generateRandomNumber(0, 1)$
5: $probValue \leftarrow B(C, P)$
6: **if** $(e^{probValue} > randNumber)$ **then**
7: **return** *true*
8: **end if**
9: **return** $false$

4.5.1. Inner search

The sequential SA algorithm is given in Algorithm 7. As we describe in the previous section, first, the temperature values are assigned and cooling rate is calculated. Then, NS generation phase and gain calculation are repeated until either all t-tuples are covered or reach to final temperature.

Algorithm 7 Simulated Annealing

Input: S: CA state, N: size of CA, N_u: number of uncovered t-tuples, t: strength, $M =< O, V, Q >$: System Model, P_0: initial temperature, P_f: final temperature
Output: S: CA state, N_u: number of uncovered t-tuple

1: $P \leftarrow P_0$
2: $R = calculateCoolingRate(k, t, N)$
3: **while** ($N_u > 0$ *and* $P > P_f$) **do**
4: $\quad S_{NS} \leftarrow generateNeighbourState(M, S)$
5: $\quad C \leftarrow fitnessFunction(S_{NS}, S, k, t)$ # C is gain
6: $\quad$ **if** ($isAccepted(C, P)$) **then**
7: $\quad\quad S \leftarrow S_{NS}$
8: $\quad\quad N_u \leftarrow N_u - C$
9: $\quad$ **end if**
10: $\quad P \leftarrow P - (P \times R)$
11: **end while**
12: **return** S, N_u

Algorithm 8 Parallel Simulated Annealing

Input: S: CA state, N: size of CA, N_u: number of uncovered t-tuples, t: strength, $M =< O, V, Q >$: System Model, P_0: initial temperature, P_f: final temperature
Output: S: CA state, N_u: number of uncovered t-tuple

1: $P \leftarrow P_0$
2: $R = calculateCoolingRate(k, t, N)$
3: $S_{NS} \leftarrow generateNeighbourState(M, S)$
4: **while** ($N_u > 0$ *and* $P > P_f$) **do**
5: $\quad NSdecisionAndGenerateNewNSKERNEL <<< B, T >>> (M, S)$
6: $\quad cudaDeviceSynchronize()$
7: $\quad fitnessKERNEL <<< B, T >>> (S_{NS}, S, k, t)$
8: $\quad cudaDeviceSynchronize()$
9: $\quad P = P - (PxR)$
10: **end while**
11: **return** S, N_u

We propose a novel approach for parallelizing SA algorithm on GPU. In contrast to parallel CCM algorithm, we need to provide a synchronization across blocks for all threads to calculate the gain. In order to decide acceptance of neighbour state, all threads has to know that counting procedure for every option combination is done , i.e., they have to be synchronized at the same point to move on the decision step. For this purpose, *cudaDeviceSynchronize*() function is used for synchronizing the GPU device with CPU i.e., every alive thread in the device is done for computing. The parallel inner search approach is given in Algorithm 7.

In the following sections how we generate neighbour state and a method to calculate gain are explained in detail.

4.5.2. Neighbour state generation

Neighbour state generation phase is done virtually, i.e., the state is not generated indeed, only the change to be done is stored in the memory. The following sections explain how sequential and parallel approaches are implemented for the NS generation.

4.5.2.1. Sequential NS generation

We provide 2 algorithms for different cases of system model e.g., SUT without constraints (Algorithm 9) and with constraints (Algorithm 10).

In Algorithm 9, we simply choose a random position in the array (lines 1-2) and find the number of settings which the corresponding option can take (line 3). Then, we find the value in that position (line 4) and choose a new value for the position (line 5). In order to prevent choosing the same value, we decrease the number of possible values by 1 and if the chosen value is equal to the position itself, we change the chosen value with the decreased value (lines 6-8).

Algorithm 9 Neighbour State Generation without Constraints

Input: $M =< O, V, Q >$: System Model, S : CA state, N : size of CA
Output: P_c : Chosen position column (option), P_r : Chosen position row,
P_s : Chosen position setting, P_{ns} : Chosen position next setting (neighbour state)

1: $P_c \leftarrow random(0, k-1)$
2: $P_r \leftarrow random(0, N-1)$
3: $N_v \leftarrow |V_i|$
4: $P_s \leftarrow S[P_r][P_c]$
5: $P_{ns} \leftarrow random(0, N_v - 2)$
6: **if** $(P_{ns} = P_s)$ **then**
7: $\quad P_{ns} = N_v - 1$
8: **end if**
9: **return** $P_c,\ P_r,\ P_s,\ P_{ns}$

The function returns the column (option) index P_c, row index P_r, position value P_s and the next position value P_{ns}.

On the other hand, if there exists any constraints between options of SUT, the new value for the randomly chosen position may violate some configurations. In Algorithm 10, after choosing a random position and a value for that position as in Algorithm 9, we also check whether the change violates any constraints (line 10). Until a valid configuration is found, new value is searched for the chosen position without cooling down the temperature (lines 2-11).

Algorithm 10 Neighbour State Generation with Constraints

Input: $M =< O, V, Q >$: System Model, S : CA state, N : size of CA
Output: P_c : Chosen position column (option), P_r : Chosen position row,
P_s : Chosen position setting, P_{ns} : Chosen position next setting (neighbour state)

1: $isViolated \leftarrow True$
2: **while** $isViolated$ **do**
3: $\quad P_c \leftarrow random(0, k-1)$
4: $\quad P_r \leftarrow random(0, N-1)$
5: $\quad N_v \leftarrow |V_i|$
6: $\quad P_{ns} \leftarrow random(0, N_v - 2)$
7: $\quad$ **if** $(P_{ns} = P_s)$ **then**
8: $\quad\quad P_{ns} = N_v - 1$
9: $\quad$ **end if**
10: $\quad isViolated \leftarrow isConstraintViolated(S, P_c, P_r, P_{ns}, Q)$
11: **end while**
12: $P_s \leftarrow S[P_r][P_c]$
13: **return** $P_c,\ P_r,\ P_s,\ P_{ns}$

4.5.2.2. Parallel NS generation

Parallel approach of neighbour state generation is very similar to sequential algorithm (Section 4.5.2.1). The only difference is that in parallel approach, every thread checks different constraint in the validation of NS. Even if only one thread can not validate its constraint, a new NS is generated and these steps are repeated until a valid configuration is found. This concurrency saves us iterating over each constraint in every SA inner loop iterations.

We do not give the parallel version of the system without constraints since it is same as sequential. The complete algorithm for the constrained systems is given as follows.

Algorithm 11 Parallel Neighbour State Generation with Constraints

Input: $M =< O, V, Q >$: System Model, S : CA state, N : size of CA
Output: P_c : Chosen position column (option), P_r : Chosen position row,
P_s : Chosen position setting, P_{ns} : Chosen position next setting (neighbour state)

shared int $P_c,\ P_r,\ P_s,\ P_{ns}$
shared bool $isViolated$
$isViolated \leftarrow True$
while $isViolated$ **do**
    if $T_{Id} = 0$ **then**
        $P_c \leftarrow random(0, k-1)$
        $P_r \leftarrow random(0, N-1)$
        $N_v \leftarrow max_{i \in [0,k-1]} |V_i|$
        $P_{ns} \leftarrow random(0, N_v - 2)$
        if $(P_{ns} = P_s)$ **then**
            $P_{ns} = N_v - 1$
        end if
    end if
    syncthreads()
    $isViolated \leftarrow isConstraintViolated(S, P_c, P_r, P_{ns}, Q)$
    syncthreads()
end while
$P_s \leftarrow S[P_r][P_c]$
return $P_c,\ P_r,\ P_s,\ P_{ns}$

4.5.3. Fitness function

In the previous sections, we explained how we generate an NS. Now, we explain our novel approach to calculate the gain of accepting the NS.

The key point of the SA algorithm is calculating the gain of NS in shorter time, since it is used in each iteration of inner search and the number of iterations is dependent on the number of t-tuples exponentially.

While calculating the gain, it is not necessary to find the number of uncovered t-tuples of both current state and NS. Because, in the NS generation phase (Section 4.5.2), only a single random position and a new value for that position are chosen. If the NS is accepted, it affects only the option combinations whose one of option is the chosen position column (fixed column or option). Based on this observation, we suggest that not all t-tuples are needed to be checked whether they are covered or not in each iteration i.e., checking only the t-tuples of those option combinations is enough for calculating gain of NS. This idea helps us to decrease the search space (number of option combinations) from $\binom{k}{t}$ to $\binom{k-1}{t-1}$ when one option is fixed in option combinations.

Moreover, we make further improvements by eliminating the unnecessary rows which do not affect the gain calculation. In NS generation (Section 4.5.2), we choose a single value for the position can take. Therefore, in that specific chosen row (configuration), we loose one covered t-tuple and generate a new one instead. However, If there is another same t-tuple in one of other configurations, while changing value of the chosen position, we actually do not loose any t-tuple since it is already covered by another configuration as well. In the same perspective, if the t-tuple which is generated by accepting the neighbour state, already exists in one of other configurations, we do not cover any new uncovered t-tuple by accepting the NS. Based on these observations, we claim that the gain is affected only from those rows which has either the chosen position value or the next value of this position in the fixed column. We only need to iterate over those rows and number of rows to be investigated decreases, significantly.

We illustrate these procedures in the following example.

Example 1 *Consider the 2-way incomplete ternary CA (each option takes 3 values* $\{0, 1, 2\}$*) without any constraints given in Figure 4.4. Assuming this state is generated in one of the SA iterations and for the NS, second row of* o_0*'s position is chosen. Since each options' column takes 3 values* $\{0, 1, 2\}$*, there are 2 remaining possible values that can be chosen for the NS generation* $\{1, 2\}$*. Let's assume that this number is 1.*

	o_0	o_1	o_2	o_3	o_4
r_0	1	2	2	0	1
r_1	**0**	0	0	2	1
r_2	2	1	2	0	1
r_3	0	2	0	1	2
r_4	0	1	0	0	2
r_5	1	0	1	0	0

(a)

	o_0	o_1	o_2	o_3	o_4
r_0	1	2	2	0	1
r_1	**1**	0	0	2	1
r_2	2	1	2	0	1
r_3	0	2	0	1	2
r_4	0	1	0	0	2
r_5	1	0	1	0	0

(b)

Figure 4.4: A 2-way CA state and a neighbour state

The current state is as given in Figure 4.4.(a) and NS is the same as current state only second row of o_0*'s column is 1 instead of 0 (Figure 4.4.(b)). In order to calculate the gain of NS, first, we generate all 2-way option combinations which contain* o_0 $\{< o_0, o_1 >, < o_0, o_2 >, < o_0, o_3 >, < o_0, o_4 >\}$*. Then, we iterate over first column* o_0 *(randomly chosen column) values and store the row indices which has either 0 or 1 except the chosen row* $\{r_0, r_3, r_4, r_5\}$*. After that, existence of every 2-tuple of chosen row* r_1 *which interacts directly to the chosen position is investigated in current state and NS* $\{(o_0 = 0, o_1 = 0), (o_0 = 0, o_2 = 0), (o_0 = 0, o_3 = 2), (o_0 = 0, o_4 = 1), (o_0 = 1, o_1 = 0), (o_0 = 1, o_2 = 0), (o_0 = 1, o_3 = 2), (o_0 = 1, o_4 = 1)\}$.

In the first iteration, all rows $\{r_0, r_3, r_4, r_5\}$ *of* (o_0, o_1) *are scanned and 2-tuples are checked for the current state* $(o_0 = 0, o_1 = 0)$ *and for the NS* $(o_0 = 1, o_1 = 0)$*. There can be 4 cases for this situation by accepting NS;*

1. *They may both* $(o_0 = 0, o_1 = 0)$ *and* $(o_0 = 1, o_1 = 0)$ *exist in another row. So, the number of uncovered 2-tuples is not changed since both of them already exist. We neither loose any covered 2-tuple nor cover any uncovered 2-tuple.*

2. *They may both* $(o_0 = 0, o_1 = 0)$ *and* $(o_0 = 1, o_1 = 0)$ *not exist in another row. So the number of uncovered 2-tuples is not changed since one 2-tuple is gone missing but instead of it, a new uncovered 2-tuple is covered.*

3. $(o_0 = 0, o_1 = 0)$ *may exist but* $(o_0 = 1, o_1 = 0)$ *may not exist in another row. 2-tuple of current state already exist so changing* $o_0 = 0$ *to* $o_0 = 1$ *does not the decrease the number of uncovered 2-tuples. However, 2-tuple of NS does not exist, so by changing the value, we cover one more 2-tuple. The number of uncovered 2-tuples decreases by 1.*

4. $(o_0 = 0, o_1 = 0)$ *may not exist but* $(o_0 = 1, o_1 = 0)$ *may exist in another row. 2-tuple of current state does not exist so changing* $o_0 = 0$ *to* $o_0 = 1$ *increases the number of uncovered 2-tuples since we loose one of the covered 2-tuples. However, 2-tuple of NS already exist, so we do not cover a new 2-tuple.*

In our example, case 4 is valid for the option combination $< o_0, o_1 >$*. If we change the value* $o_0 = 0$ *to* $o_0 = 1$*, the 2-tuple* $(o_0 = 0, o_1 = 0)$ *goes missing i.e., this t-tuple is not covered by any other rows. However,* $(o_0 = 1, o_1 = 0)$ *is already covered by last row, so we do not cover any uncovered 2-tuple. The number of 2-tuples is increased by 1 for the option combination* $< o_0, o_1 >$*. Same procedure is done for every option combination.*

4.5.3.1. Sequential fitness function

In the previous section, we explained the general idea of how fitness function is operated. In this section, we describe the sequential implementation of fitness function to calculate the gain.

The complete algorithm for fitness function is given in Algorithm 12. In the first loop (lines 3-7), necessary rows are collected in a set as we describe in the previous section. In the next loop (lines 9-26), option combinations which contain the fixed option are generated iteratively and for each option combination number of uncovered t-tuples is counted. The variables $existBefore$ and $existAfter$ stand for whether the t-tuple of current state and NS of the corresponding option combination in the chosen row exist in another row or not. 4 cases that we describe in the Example 1 occur with respect to these variables. Since only 2 of the cases affect the gain, we only check these cases (lines 21-25).

Algorithm 12 Fitness Function

Input: N: size of CA, P_c: Chosen position column (option), P_r: Chosen position row, P_s: Chosen position setting, P_{ns}: Chosen position next setting (neighbour state), S : CA state, k : Number of options, t : strength
Output: C : gain

```
1:  C ← 0
2:  rowIndices ← {}
3:  for (r ← [0, ..., N − 1]) do
4:      if ( (S[r][P_c] = P_s or S[r][P_c] = P_ns) and (P_r ≠ r) ) then
5:          rowIndices.append(r)
6:      end if
7:  end for
8:
9:  for (i ← [0, ..., N_optComb − 1]) do
10:     optComb_i ← generateOptCombFromIndex(k, t, i)
11:     for (r in rowIndices) do
12:         if (S[r][∀o_j ∈ optComb_i] = S[P_r][∀o_j ∈ optComb_i]) then
13:             existBefore ← true
14:         else
15:             existAfter ← true
16:         end if
17:         if (existAfter and existBefore) then
18:             break
19:         end if
20:     end for
21:     if (!existAfter and existBefore) then
22:         C ← C + 1
23:     else if (existAfter and !existBefore) then
24:         C ← C − 1
25:     end if
26: end for
27: return C
```

4.5.3.2. Parallel fitness function

As we mentioned in Section 4.5.3, the crucial part of the SA algorithm is the fitness function where the gain of the NS is calculated. Calculating the gain is actually counting the number of t-tuples over and over again. That's why parallel computing techniques carry big importance in this part.

Before describing the algorithm in detail, we explain how we manage the resources of GPU. As in CCM, we use 32 blocks and 1024 threads for each block. In Section 4.5.3, we explained how to remove the unnecessary rows to avoid iterating them. Based on this idea, we claim that sometimes, $|V_{P_c}|$ can be high for randomly chosen option P_c in NS. Since option values from V_{P_c} are distributed to the column almost equally sized, number of unnecessary rows may become high. Because, we only concern about the rows which have either the chosen column setting P_s or the next setting P_{ns}. Depending on this idea, threads within the same warps are divided into 4 groups to get full performance of GPU and all groups across blocks are indexed using (4.8). Otherwise, in the cases of having less number of necessary rows may cause some of the threads in a warp (sometimes more than half) waited i.e., they do not execute any row since there are more than enough threads for the rows.

$$G_{Id} = \frac{(1024 \times N_B) + T_{Id}}{8} \tag{4.8}$$

Number of groups becomes $4 \times N_w$. However, if this number is not enough, then each group takes more than 1 option combination and process them iteratively.

$$N_w = \frac{1024}{32} \times N_B \tag{4.9}$$

After that, each option combination is assigned to a group with respect to group index. In order to use warp synchronization, it is important that number of uncovered t-tuples of any option combination is counted only by the threads within the same warp.

The complete algorithm is given as follows (Algorithm 13)

Algorithm 13 Parallel Fitness Function

Input: N: size of CA, P_c: Chosen position column (option), P_r: Chosen position row, P_s: Chosen position setting, P_{ns}: Chosen position next setting (neighbour state), S : CA state

Output: C : gain

1: $C \leftarrow 0$
2: $rowIndicesTemp \leftarrow \{false\}$
3: $r \leftarrow T_{Id}$
4: **while** $r < N$ **do**
5: **if** ($(S[r][P_c] = P_s \; or \; S[r][P_c] = P_{ns}) \; and \; (P_r \neq r)$) **then**
6: $rowIndicesTemp[r] \leftarrow true$
7: **end if**
8: $r \leftarrow r + blockDim.x$
9: **end while**
10: ***synchronize()***
11: $rowIndices \leftarrow prefixSum(rowIndicesTemp, N)$
12: ***synchronize()***
13:
14: $existBefore \leftarrow \{false\}$
15: $existAfter \leftarrow \{false\}$
16: **while** $(Id < N_{optComb} - 1)$ **do**
17: $optComb_{Id} \leftarrow generateOptCombFromIndex(G_{Id})$
18: $ind \leftarrow T_{Id} \; (mod \; 8)$
19: **while** $(ind < length(rowIndices))$ **do**
20: $r \leftarrow rowIndices[ind]$
21: **if** $(S[r][\forall o_j \in optComb_{Id}] = S[P_r][\forall o_j \in optComb_{Id}])$ **then**
22: $existBefore[G_{Id}] \leftarrow true$
23: **else**
24: $existAfter[G_{Id}] \leftarrow true$
25: **end if**
26: **if** $(existAfter[G_{Id}] \; and \; existBefore[G_{Id}])$ **then**
27: **break**
28: **end if**
29: $ind \leftarrow ind + 8$
30: **end while**
31: **if** $(!existAfter[G_{Id}] \; and \; existBefore[G_{Id}])$ **then**
32: $C \leftarrow C + 1$
33: **else if** $(existAfter[G_{Id}] \; and \; !existBefore[G_{Id}])$ **then**
34: $C \leftarrow C - 1$
35: **end if**
36: $Id \leftarrow Id + (4 \times N_w)$
37: **end while**
38: **return** C

There are some improvements in parallel fitness function (Algorithm 13) compared to sequential one (Algorithm 12) in calculation time. Necessary rows are identified in parallel and their indices are stored in an array ($rowIndicesTemp$). Every thread checks a single element in the chosen option column P_c of CA state whether the value is equal to either P_s or P_{ns} as well as ensuring that the chosen row P_r is not included in the necessary row list (line 5). Every thread marks its position *true* in $rowIndicesTemp$ if the corresponding row is a necessary row, otherwise it remains as $false$ (line 6). If the number of threads in a block is not enough to scan all of the values in the column P_c, some (or all) of the threads are sent to do the same procedures again until all values are checked (line 8). Then, we use prefix sum algorithm [28] which simply puts in a order the index of the *true* values in $rowIndicesTemp$ array (line 11).

Another improvement is done in counting the missing t-tuples of option combinations i.e., counting procedure is carried out in parallel. Every group of threads generate their own option combinations based on their G_{Id} (line 17) and if the number of group is less than number of option combinations, then every group takes more than 1 option combination iteratively (line 36). Every thread within the same group is indexed to a number between 0 and 7 since there are 8 threads in each group (line 18). These 8 threads scan the option combination columns of CA to check whether the t-tuple already exists before accepting NS or after accepting NS (lines 21-25). Rows are distributed to the threads based on their *ind* values (line 29).

The rest of the algorithm is same as the sequential algorithm. There are 4 cases that needs to be considered but only 2 of them make difference in gain (lines 31-35).

4.6. Multiple Neighbour States Generation in Parallel

So far, we give algorithms explaining how to parallelize SA functions for GPU-based parallel computing techniques. These algorithms improve the efficiency in time but not in quality. They actually do the same things with sequential algorithm however, with improved methods. In this section, we present an algorithm to generate multiple NSs in parallel in order to decrease the CA construction time further, and to improve the quality, i.e., decrease the number of configurations as well.

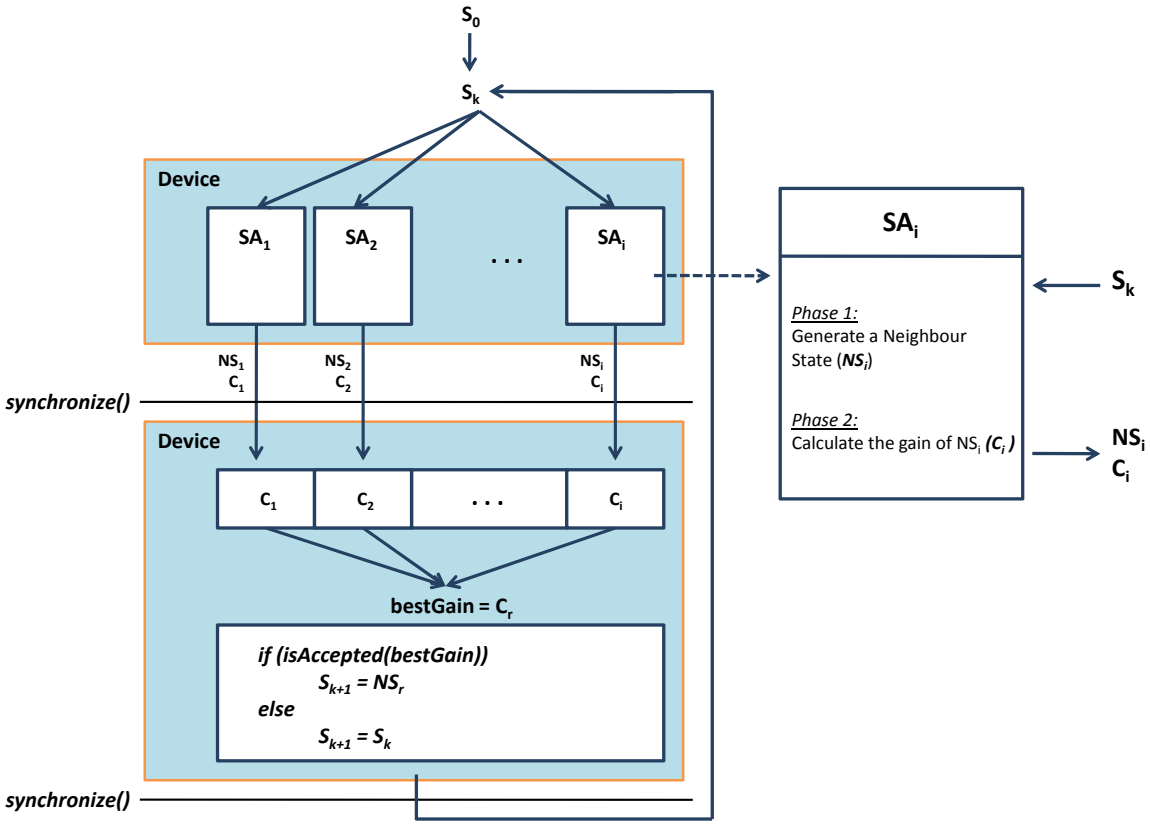

Figure 4.5: Multiple NSs generation strategy in parallel

The procedure to generate multiple NSs is given in Figure 4.5. First, the initial state S_0 is generated (Section 4.3) and provided to the system. Then, every SA algorithm takes the same S_0 as an input and generate different NSs. Each SA calculates gain of its own NS. After all SA algorithms are done with NS generation and gain calculation (synchronize),

they return the NSs and gains as outputs. Then, maximum gain is chosen among all gains and assigned to *bestGain*. This *bestGain* is fed into decision function to accept or reject the corresponding NS. If NS is accepted, then it is assigned to next state S_{k+1}, otherwise, S_k is assigned. This steps are repeated until reaching the final temperature or constructing a complete CA.

The outer search algorithm is same as with Algorithm 1, only the simulated annealing function is as Figure 4.5.

4.7. Hybrid Approach

While conducting experiments, we realize that, sometimes inner search algorithms may return very small number of uncovered t-tuples. Only for those t-tuples, size of CA is increased and extra 1 or 2 more SA runs are executed. Even though, sometimes, additional configurations (size increment) may not be enough to construct a complete CA and the number of SA runs are increased more than enough. Especially, when the configuration space is large and constraints are very dense, this situation is more prone to occur. Only for small number of remaining t-tuples, these extra SA runs may not be affordable since SA takes more time when the configuration space is large and approach may suffer both in time and size. Hence, decreasing number of SA runs as much as possible while keeping the quality same or further improving, is very important. Based on these observations, we propose a novel hybrid approach using GPU-based parallel computing techniques and SAT solver. We claim that when the number of t-tuples is small enough, SAT solver can be a very efficient way to cover those remaining t-tuples by generating additional configurations. This technique may help us constructing CAs faster and cheaper.

Algorithm 14 Generating Additional Configurations

Input $M =< O, V, Q >$: SUT Model, $uncoveredTuples$: uncovered t-tuples,
Output Configurations which cover given uncovered t-tuples

1: $N_{r_{extra}} \leftarrow estimate(uncoveredTuples)$
2: $isSatisfied \leftarrow false$
3: **while** $!isSatisfied$ **do**
4: $\quad isSatisfied = isSatisfiable(M, N_{r_{extra}}, uncoveredTuples)$
5: $\quad N_{r_{extra}} \leftarrow N_{r_{extra}} + 1$
6: **end while**

In Algorithm 14, first we estimate the minimum number of configurations $N_{r_{extra}}$ by identifying the option whose different settings exist more in the remaining t-tuples. Since a single row can take only a single value for a specific option, there needs to be at least 1 different row for each different option values. After that, we simply pass uncovered t-tuples and $N_{r_{extra}}$ value to the SAT solver and leave it to decide whether those remaining t-tuples can be covered by $N_{r_{extra}}$ rows or not. If it returns as $unsatisfiable$, $N_{r_{extra}}$ is increased by 1 and pass to SAT solver again. These steps are repeated until SAT solver returns a $satisfiable$ solution.

We use a simple encoding technique for SAT solver [3]. First, we define the domain values for each cell a_{rc} in the array $N_{r_{extra}} \times k$ where r is the row (configuration) index and c is the column (option) index of a_{rc}. Then, we define rules for each constraint to make sure that each configuration is valid. In order to do that, we simply add negation of constraints for each row. In the final step, we define rules to cover remaining uncovered t-tuples. We illustrate these procedures in the following example;

Example 2 *Consider that a SUT has 5 options* $\{o_0, o_1, o_2, o_3, o_4\}$ *and these options take 3, 4, 3, 4 and 2 values, respectively. Also, 2 invalid t-tuples exists within the system;* $\{(o_0 = 0, o_1 = 0), (o_2 = 2, o_4 = 1)\}$. *Assume that the inner search approach is operated on this configuration space model and these remaining t-tuples are left uncovered:* $\{(o_3 = 0, o_4 = 1), (o_2 = 1, o_3 = 0)\}$. *We try to cover these t-tuples with 2 configurations.*

To do so, we need to define all possible values for each cell of the 2×5 *additional configuration array. For instance, as in the example, first row and first column of the array* a_{00} *can take 3 values* $\{0, 1, 2\}$.

After defining all cells in this format, each constraint is needed to be introduced to the SAT solver to ensure that each row is a valid configuration. For example, the invalid t-tuple !(o_0 = 0 and o_1 = 0) is defined for each row separately, using the conjunction "and". Then, invalid t-tuple rule is simply defined by making at least one of the option value is different than invalid t-tuple.

Uncovered t-tuples is defined similarly with constraints. In the constraint definition, we change the "and" statement with "or", since it is enough for the t-tuple to be covered at least one row. Also, we change all "or" statements to "and" statement and change the non-equal statement "ne" to equality "eq" statement to make sure of that each option take the right value.

The complete procedure for the proposed approach is given in Algorithm 15.

SAT Solver Encoding Example

Option values
(int a_{00} 0 2)
(int a_{10} 0 2)
(int a_{01} 0 3)
(int a_{11} 0 3)
(int a_{02} 0 2)
(int a_{12} 0 2)
(int a_{03} 0 3)
(int a_{13} 0 3)
(int a_{04} 0 1)
(int a_{14} 0 1)

Constraints
(and (or (ne a_{00} 0) (ne a_{01} 0)) (or (ne a_{10} 0) (ne a_{11} 0))) # !(o_0 = 0 and o_1 = 0)
(and (or (ne a_{02} 2) (ne a_{04} 1)) (or (ne a_{12} 2) (ne a_{14} 1))) # !(o_2 = 2 and o_4 = 1)

Uncovered t-tuples
(or (and (eq a_{03} 0) (eq a_{04} 1)) (and (eq a_{13} 0) (eq a_{14} 1))) # (o_3 = 0 and o_4 = 1)
(or (and (eq a_{02} 1) (eq a_{03} 0)) (and (eq a_{12} 1) (eq a_{13} 0))) # (o_2 = 1 and o_3 = 0)

Algorithm 15 Covering Array Generation (hybrid approach)

Input: $M =< O, V, Q >$: SUT Model, t: strength, P_0: initial temperature, P_f: final temperature

Output: $CA(N; k, t, M =< O, V, Q >)$: t-way Covering Array

1: $B_l \leftarrow 0,\ B_u \leftarrow INT\_MAX$
2: $isLowBoundFound \leftarrow false$
3: $isUpBoundFound \leftarrow false$
4: $T \leftarrow k$
5: $N \leftarrow estimateLowerBound(t, M)$
6: $S_0 \leftarrow generateInitialState(N, M)$
7: $N_u \leftarrow combinatorialCoverageMeasurement(S_0, t, M)$
8: $S \leftarrow S_0$
9: **while** (*true*) **do** # *part*1
10: $S, N_u \leftarrow simulatedAnnealing(S, N, N_u, t, M, P_0, P_f)$
11: **if** ($N_u > 0$ *and* $N_u < T$) **then**
12: $isLowBoundFound \leftarrow true$
13: $B_l \leftarrow N$
14: $S_l \leftarrow updateBoundaryState(B_l, S)$
15: $S_{extra} \leftarrow generateExtraOptionConfigs(S, N_u, M)$
16: **if** ($!isUpBoundFound$ *or* $B_u > N + sizeof(S_{extra})$) **then**
17: $S \leftarrow S + S_{extra}$
18: $N \leftarrow sizeof(S)$
19: $isUpBoundFound \leftarrow true$
20: $B_u \leftarrow N$
21: $S_u \leftarrow updateBoundaryState(B_u, S)$
22: $N_u \leftarrow 0$
23: **end if**
24: **break**
25: **else**
26: **if** ($N_u = 0$) **then**
27: $B_u \leftarrow N$
28: $S_u \leftarrow updateBoundaryState(B_u, S)$
29: $isUpBoundFound \leftarrow true$
30: **else**
31: $B_l \leftarrow N$
32: $S_l \leftarrow updateBoundaryState(B_l, S)$
33: $isLowBoundFound \leftarrow true$
34: **end if**
35: **end if**
36: **if** ($N_u \geq threshold$ *or* $N_u = 0$) **then**
37: $N \leftarrow nextSizePart1(N, B_u, B_l, N_u)$
38: $S \leftarrow updateCurrentState(S_l, N, N_u, isLowBoundFound)$
39: **end if**
40: **end while**

41: **while** $(B_u - B_l > 1)$ **do** # *part*2
42: $sizeCA \leftarrow nextSizePart2(B_u, B_l)$
43: $S \leftarrow updateCurrentState(S_l, N, isLowBoundFound)$
44: $N_u \leftarrow calculateNumberOfUncoveredTuples(S, t, M)$
45: $S, N_u \leftarrow simulatedAnnealing(S, N, N_u, t, M, P_0, P_f)$
46: **if** $(N_u = 0)$ **then**
47: $B_u \leftarrow sizeCA$
48: $S_u \leftarrow updateBoundaryState(B_u, S)$
49: **else**
50: $B_l \leftarrow sizeCA$
51: $S_l \leftarrow updateBoundaryState(B_l, S)$
52: **end if**
53: **end while**
54: **return** S_u

Additional to the previously described methods, in this approach (Algorithm 15), we decide a threshold for the number of uncovered t-tuples returned from SA algorithm. If it is smaller than this threshold, we use the SAT solver approach to cover these remaining t-tuples with additional configurations.

We divide the outer search into two parts. In the first part (lines 9-40), approach tries to generate a good lower boundary state whose number of uncovered t-tuples is smaller than threshold. Once the lower bound is found, additional configurations are generated to cover remaining t-tuples and are appended to CA. Assumption in here is that, the number of additional configurations is very small. So that the complete CA which is constructed by appending newly generated configurations, is an upper boundary state whose size is closer to size of lower boundary state. Therefore, the gap between lower and upper bounds becomes very small.

In the second part (lines 41-53), outer search proceeds with a binary search until a complete CA is constructed. The difference between first and second part is that first part tries to compute a good lower boundary state and chose its NS state with respect to this idea (line 37). However, in second part, only goal is to decrease the gap between upper and lower bound more until reaching to minimal CA size. So, NS size is decided according to binary search (line 42).

5

EXPERIMENTS

This chapter presents the experimental results to evaluate all proposed parallel algorithms and to make comparisons with well known existing tools.

The sequential codes are executed on a CentOS 6.5 high performance computing machine with Intel(R) Xeon(R) CPU 2.80GHz having total 252 GB ram. Parallel algorithms were employed on an NVIDIA Tesla K40 graphics accelerator with 2880 CUDA cores.

Different types of experiments have been conducted in this chapter, therefore the details of each experimental setup, evaluation framework, data analysis and discussions of experiments are given separately in different sections.

5.1. Experiments on Combinatorial Coverage Measurement

In this set of experiments, we present experimental results of both sequential (Section 4.4.1) and parallel (Section 4.4.2) implementations of CCM algorithm. We provided randomly generated CAs to the algorithms and measure the execution times of the experiments.

5.1.1. Experimental setup

We generated 20 non-complete CAs randomly for each configuration space with $t \in \{2, 3, 4, 5\}$ and $k \in \{40, 80, 120, 160, 200, 400, 600, 800, 1000, 1500\}$ each option having either 2, 3, or 4 settings since they are mostly small numbers in practical scenarios. We chose the size of CAs close to their approximated sizes as 32, 128, 1280 and 8096 for strength 2, 3, 4 and 5, respectively. In total, 1080 experiments have been conducted in this section.

Since these comparisons are not dependent on the number of constraints, we assumed that all configuration spaces are constraint free. Because, in both parallel and sequential CCM algorithms, we count the number of all invalid t-tuples in the beginning of the algorithm using same sequential method, and after that we proceed with the counting approach like no constraint exists in the system, i.e., we count every t-tuple including invalid t-tuples.

5.1.2. Evaluation framework

We evaluated the parallel CCM algorithm's execution time by changing the number of options and keeping the strength fixed.

5.1.3. Results and analysis

All presented execution time results are averages of 20 experiments which were conducted on different configuration spaces but with the same system specification.

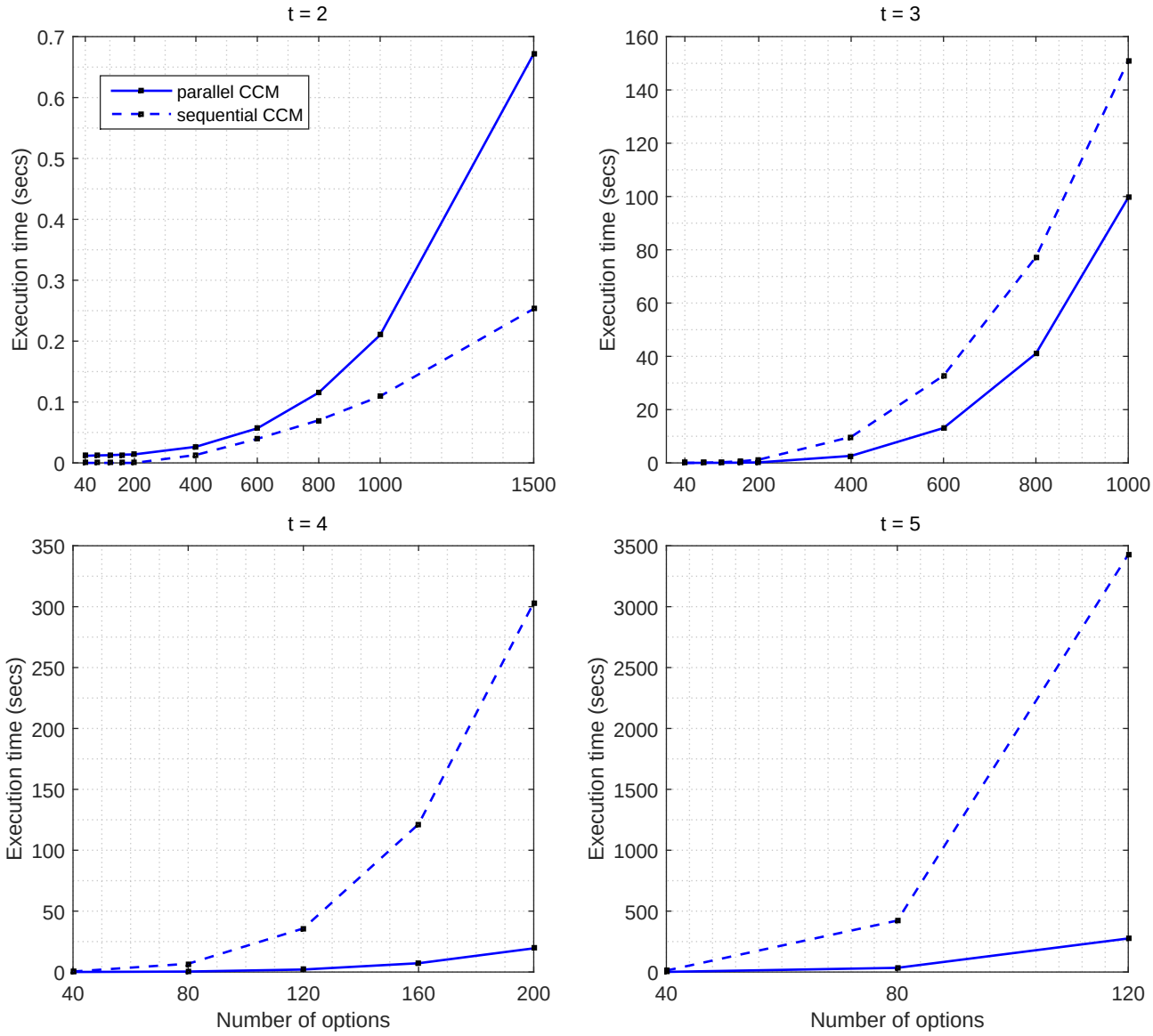

Figure 5.1: Comparing execution time results of parallel and sequential CCM algorithms for t=2, 3, 4 and 5

Figure 5.1 makes comparisons between parallel and sequential algorithms of CCM. The sequential algorithm does better only when t=2 but not more than 0.5 secs. However, as the strength value increases, the parallel algorithm surpass the sequential algorithm, significantly. Based on these results, we can claim that as the strength value increases, efficiency of parallel approach gets more dominant.

5.1.4. Discussions

We previously argued that combinatorial problems were actually simple counting techniques. In here, this conjecture can be justified.

When t=2, since the configuration space is not large and the size of CA is small, the number of t-tuples to be counted is small, too. As the strength value and the size increases, the number of t-tuples increases, exponentially. Therefore, our conjecture holds in these cases and encourages us to move forward with this work further to construct CAs using GPU-based algorithms.

5.2. Experiments on Simulated Annealing

In this section, we present experimental results of both sequential and parallel SA algorithms (Section 4.5) to construct CAs. In these experiments, a single SA algorithm was executed to see the time efficiency of the parallel algorithm over the sequential one.

We provided the size of CA to the algorithm and made sure that no complete CA exists with the given size. Because, it is important that SA cools down from initial temperature to final temperature without constructing a complete CA, i.e., without breaking the loop. For that purpose, we simply assign 75% of the ACTS results as CA sizes.

5.2.1. Experimental setup

In these experiments, we generated 5 different configuration spaces for each system specifications with $t \in \{2, 3\}$, $k \in \{20, 40, 60, 80, 100, 120, 140, 160, 180, 200\}$ and $0 \leq |Q_i| \leq 9$. Each settings of options was assigned to 2, 3, or 4 randomly. For each number of options, 10 level constraint sizes were calculated using (5.1) and all constraints were generated randomly. We assumed that all constraints are invalid t-tuples with length 2 or 3 and no more invalid t-tuple exists beside them.

$$|Q_i| = \lfloor 2i\sqrt{k} \rfloor,\ 0 \leq i \leq 9 \tag{5.1}$$

In total, we have conducted 2000 experiments for this section.

$$\{2 \times algos\} \times \{2 \times t\} \times \{10 \times k\} \times \{10 \times Q_i\} \times \{5 \times conf.\ space\} = 2000\ experiments$$

In order to keep the decrement of number of valid configurations as linear as possible in the increasing number of constraints, we chose to use a logarithmic incremental method proportional to k for constraint sizes. The number of all valid configurations decreases exponentially with the number of constraints $|Q_i|$.

5.2.2. Evaluation framework

We evaluated our proposed algorithm over sequential one in execution time. We investigated the effectiveness of parallelism on this algorithm in 3 ways by keeping the strength always fixed and: (1) making overall comparisons, (2) keeping number of option fixed and (3) keeping constraint size fixed.

5.2.3. Results and analysis

All given results in this section are the averages of 5 experiments which were conducted on different configuration spaces having the same system specification.

Figure 5.2 illustrates all results including all constraint levels and the number of options together. For better visualization to distinguish the efficiency of parallel algorithms for different strength values, Figure 5.2.(a) shows results only for t=2 and Figure 5.2.(b) shows for t=3.

For t=2, sequential algorithm is faster in execution time in all the experiments but not more than 2 seconds. Since there are not much t-tuples to be counted when t=2, we can not benefit from GPU device with full capacity. Therefore, this result is not surprising. On the other hand, when t=3, number of t-tuples increases exponentially and the power of parallelism shows its effectiveness clearly as it can be seen in Figure 5.2.(b). The parallel SA algorithm completed the cooling down process for all experiments under 25 seconds but sequential SA algorithm reaches almost 4000 seconds in some of experiments when t=3.

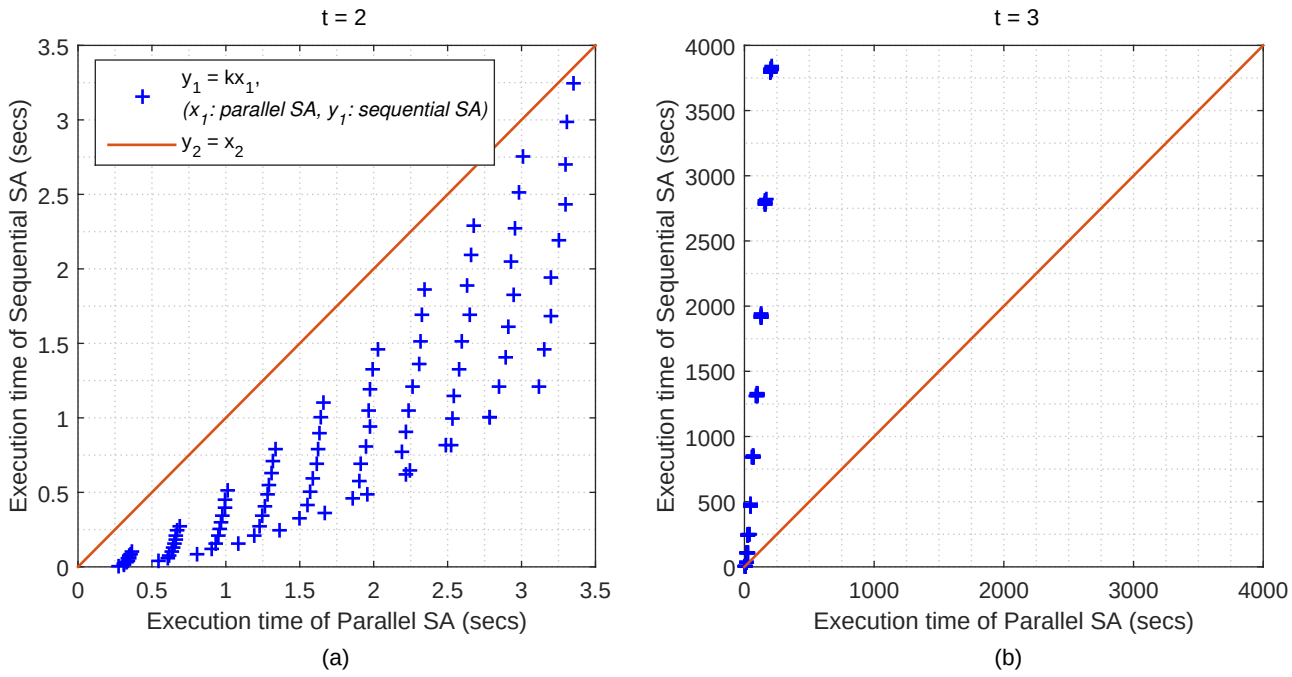

Figure 5.2: Comparing execution times of parallel and sequential SA algorithms for t=2 and t=3

Figure 5.3 illustrates the same results by keeping number of options fixed to analyse the effect of number of constraints on algorithms execution time. We presents the results for $k \in \{20, 80, 140, 200\}$ and $t \in \{2, 3\}$.

While the number of options increases for t=2 in Figure 5.3.(a), the gap between execution times of algorithms decreases as well. On the other hand, the effect of constraints is not obvious for t=3 in Figure 5.3.(b) since overall speed up is not good enough for both algorithms except when k=20.

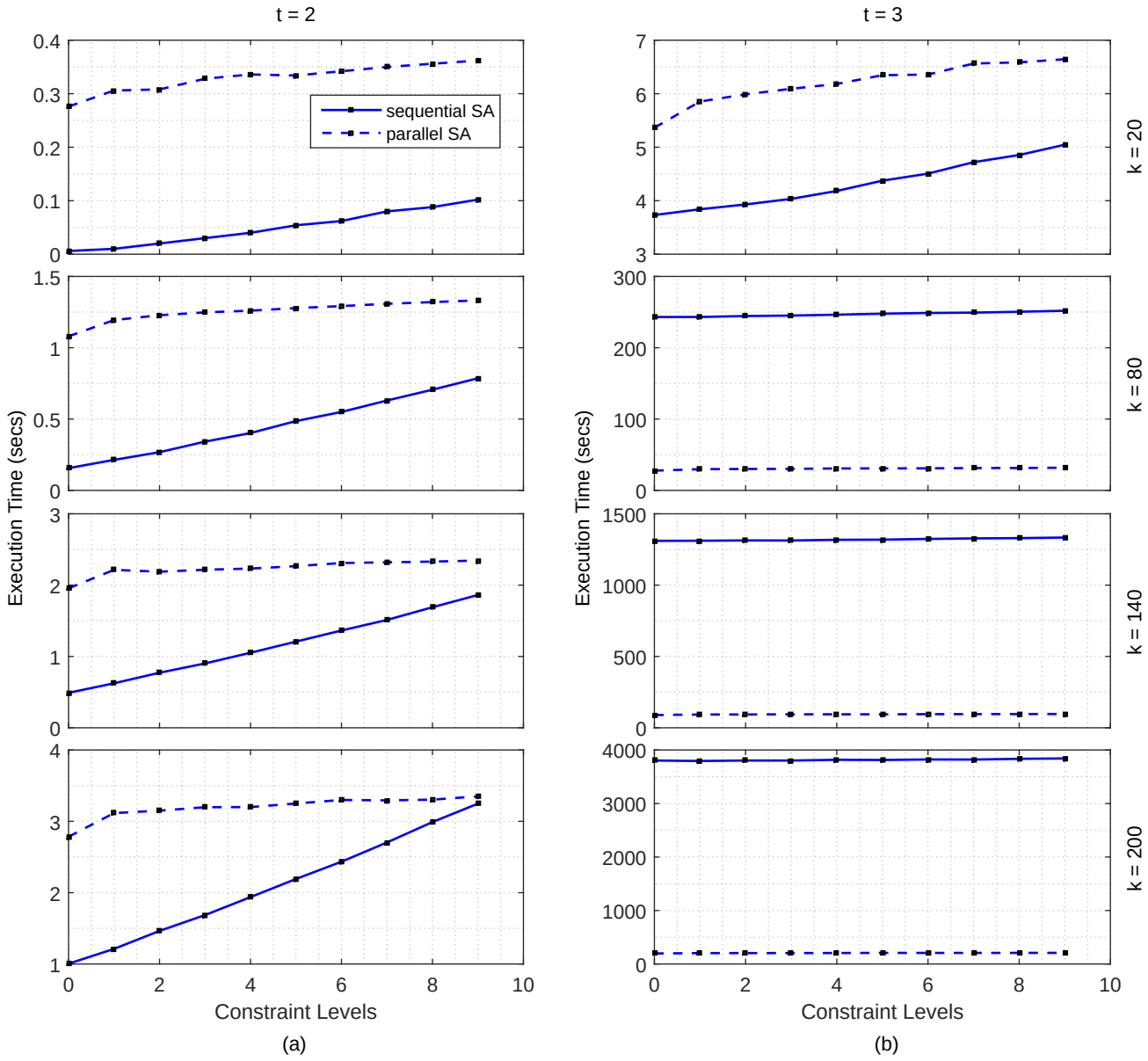

Figure 5.3: Comparing execution times of parallel and sequential SA algorithms for t=2 and t=3 when number of options is fixed

In Figure 5.4, results are presented by keeping the number of constraints fixed in order to observe the behaviours of algorithms while the number of options changes. As it can be seen in Figure 5.4.(a), when t=2, the trends of algorithms are close to each other. However, when t=3 in Figure 5.4.(b), the efficiency of the parallel algorithm dominates the sequential algorithm, noticeably. Therefore, we may comment out that increment of number of options effects the sequential algorithm in a negative way more than parallel algorithm when t=3.

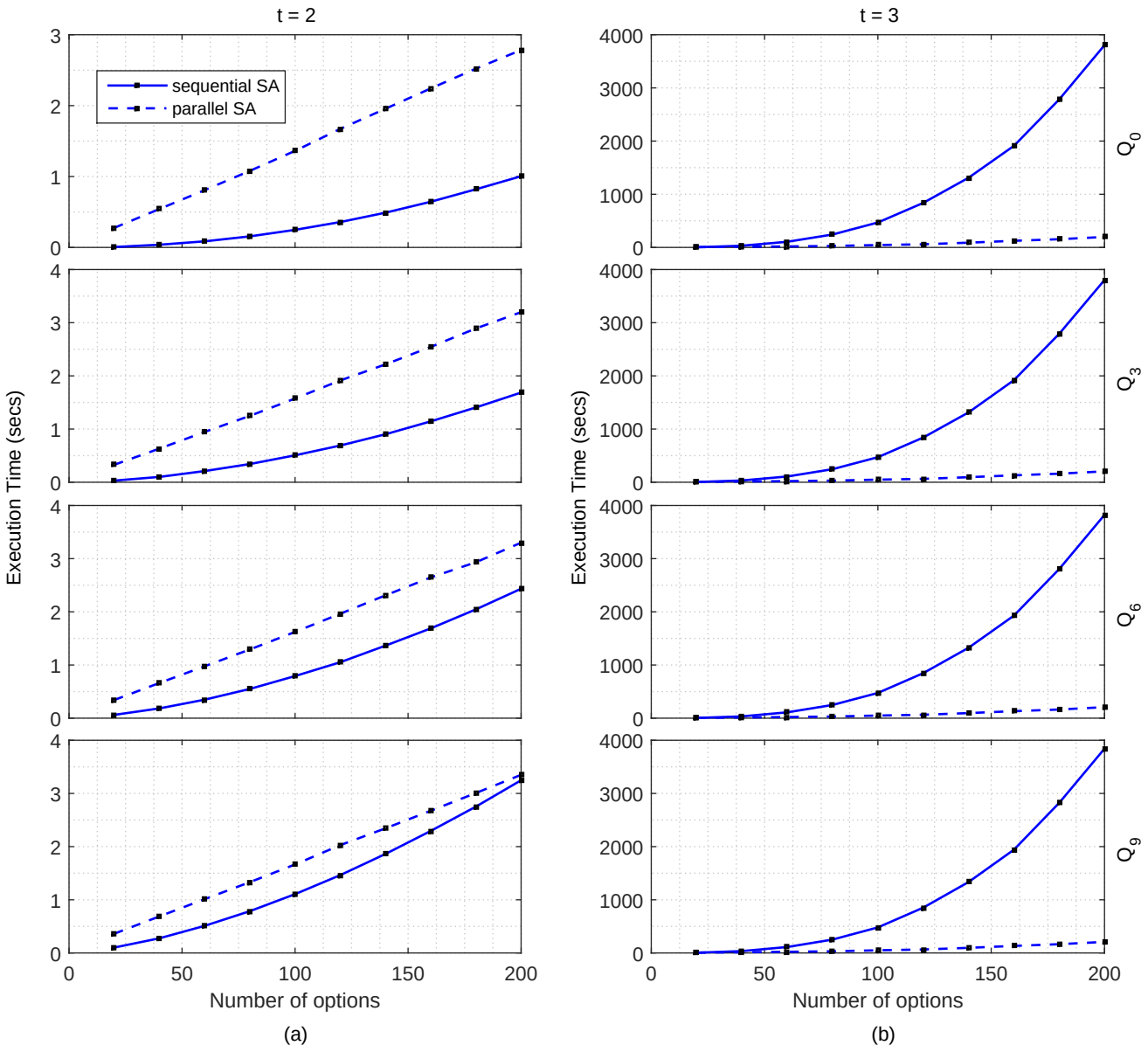

Figure 5.4: Comparing execution times of parallel and sequential SA algorithms for t=2 and t=3 when number of constraints (Q_i) is fixed

5.2.4. Discussions

Even though sequential algorithm is better than parallel algorithm when t=2, the time difference between algorithms is not more than 2 seconds. Besides that, the dominance of parallel algorithm is obvious in all Figures. There is no clear behaviours of number of constraint effects, but increment of number of options effects sequential algorithm, significantly. Therefore, we conclude that parallel SA algorithm is better than the sequential SA algorithm in execution time since the case for t=2 is negligible.

5.3. Experiments on Multiple Neighbour States Generation Strategy

In these experiments, we executed our proposed approach that is to generate multiple NSs in parallel with different number of NSs to find out the best scenario. In total, we had 32 blocks which can be used fully parallel. We used 5 different combinations, $r \times s$ where r is the number of NSs and s is the number of blocks for each gain calculation of NS: 2x16, 4x8, 8x4, 16x2, and 32x1. Then, we compare the best combination with single NS technique algorithm (1x32). Every algorithm attempted to construct a complete CA with minimal size and all procedures are included in the execution time. The algorithms used same outer search (Algorithm 4.2) for the experiments.

5.3.1. Experimental setup

In this part of experiments, we used the same configuration space with the previous section (Section 5.2.1) but with $k \in \{20, 40, 60, 80, 100\}$. In total, 2000 experiments have been conducted for this comparison.

For finding the best combination of $r \times s$:

$$\{5 \times combs\} \times \{2 \times t\} \times \{5 \times k\} \times \{4 \times Q_i\} \times \{5 \times conf.\ space\} = 1000\ experiments$$

Comparing the best combination with single NS generation technique:

$$\{2 \times combs\} \times \{2 \times t\} \times \{5 \times k\} \times \{10 \times Q_i\} \times \{5 \times conf.\ space\} = 1000\ experiments$$

5.3.2. Evaluation framework

First, we compared all $r \times s$ combinations to see how much trade off can be made between time and size. Then, we evaluated the efficiency of using multiple NSs strategy both in time and size. To do so, results are presented in several ways e.g., all results together, keeping the number of constraints or options fixed.

5.3.3. Results and analysis

Each result presented in this section is the average of 5 experimental results.

Figure 5.5 gives the experimental results of using multiple NSs strategy with different $r \times s$ combinations. As the graphs suggest that there is no combination which is the best both in time and size. If one needs the minimal size of CA, the combination of 32x1 seems the most appropriate one, however it is the worst one in execution time. On the other hand, if one needs a faster algorithm, 2x16 seems the most appropriate one among all those combinations. To sum up, there is no superior combination which is the best both in time and size. Therefore, in this case, we leave the choice for the number of NSs to the developer and choose the 4x8 system for our next experiments since the execution time results are very close to minimal one and size results are acceptable.

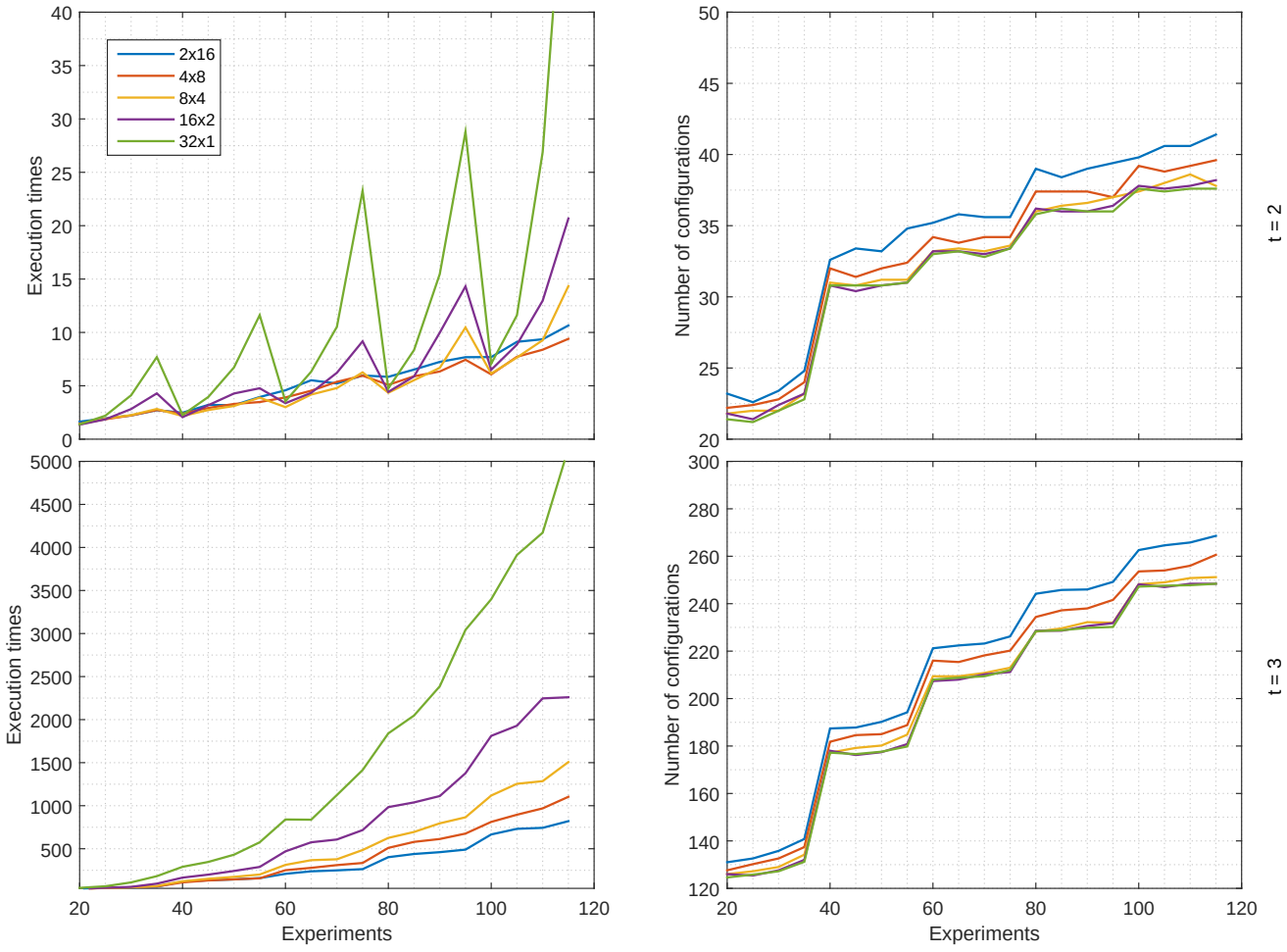

Figure 5.5: Comparing execution times and size results of 2x16, 4x8, 8x4, 16x2 and 32x1 systems

In the next part of experiments, we compare using multiple NSs strategy (4x8) with single NS strategy (1x32). Figure 5.6 illustrates the number of configuration results of the algorithms. Both strength, for t=2 in Figure 5.6.(a) and t=3 in Figure 5.6.(b), clearly indicate that multiple NSs technique constructs CAs better in size.

Figure 5.7 illustrates the execution times of the algorithms. When t=2 in Figure 5.7.(a), multiple NSs algorithm produces CAs faster than single NS algorithm. On the contrary, when t=3 in Figure 5.7.(b), single NS algorithm is much faster than multiple NSs algorithm. In this comparison, we believe that single NS algorithm dominates multiple NSs algorithm since the case for t=2 is negligible.

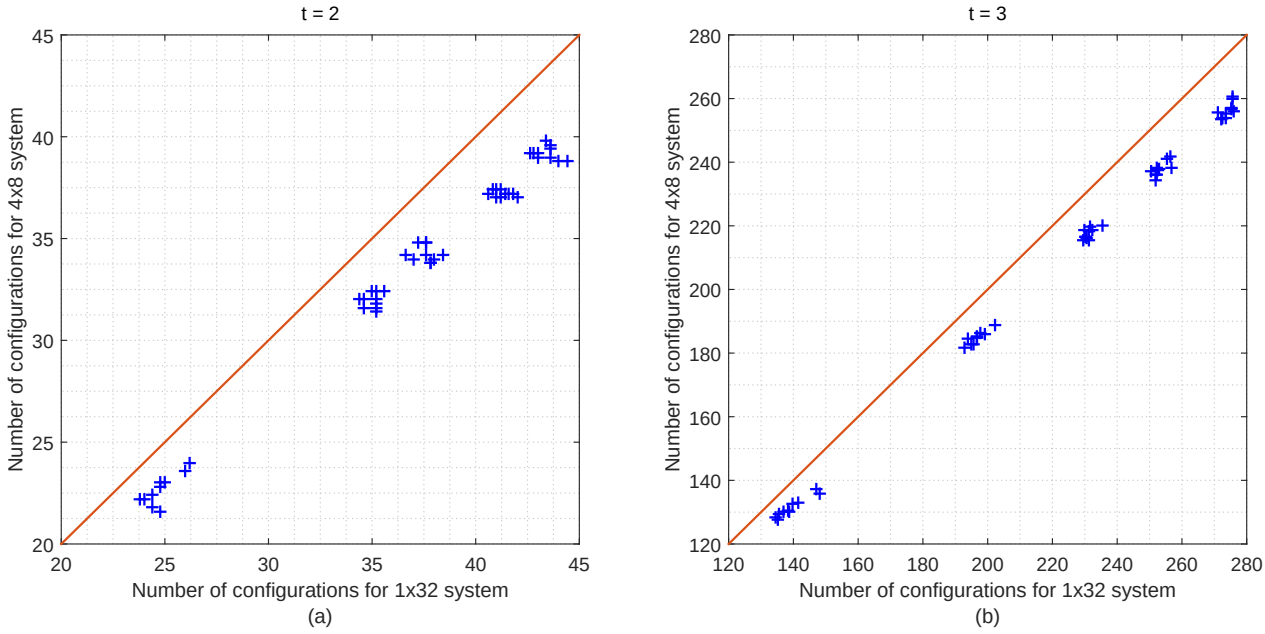

Figure 5.6: Comparing size results of 1x32 and 4x8 systems for t=2 and t=3

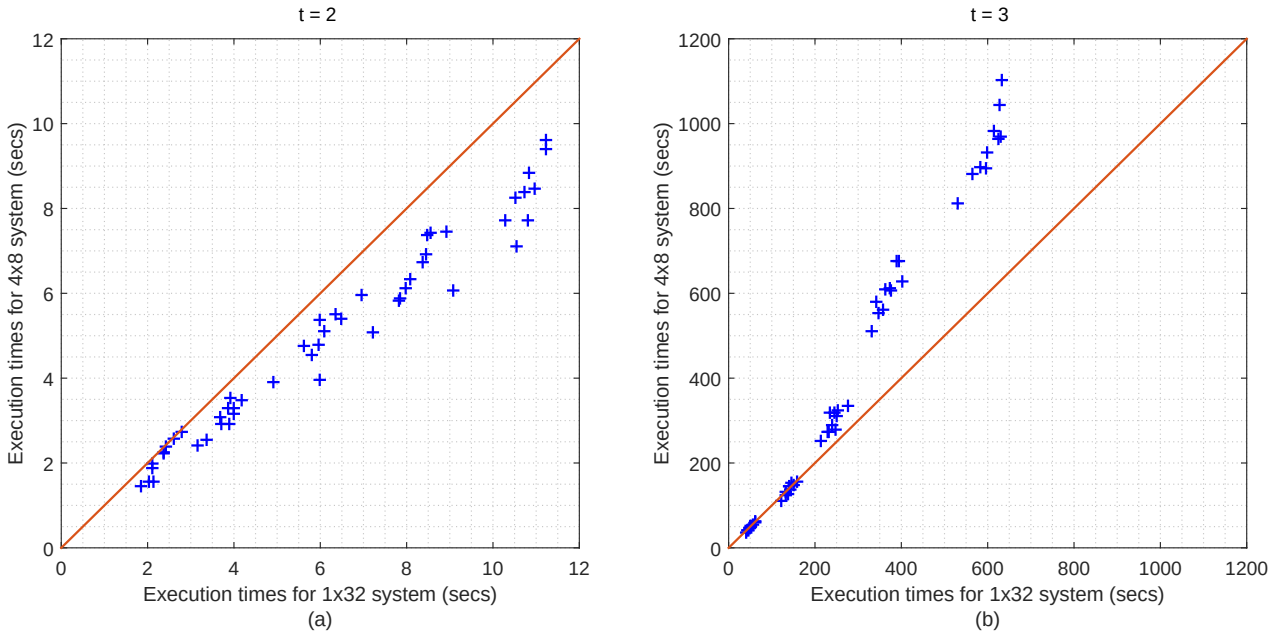

Figure 5.7: Comparing execution time results of 1x32 and 4x8 systems for t=2 and t=3

Figure 5.8 and Figure 5.9 give results of both approaches while keeping number of constraints fixed where the t=2 and t=3, respectively. Each line of graphics represents different 5 constraint levels starting form 0 to 8. Left column of figures gives the execution times of algorithms and right column presents the number of configurations of CA.

There are no obvious superiority for an algorithm over another in the increasing number of options as it can be seen from figures. The behaviours are almost same in all the cases.

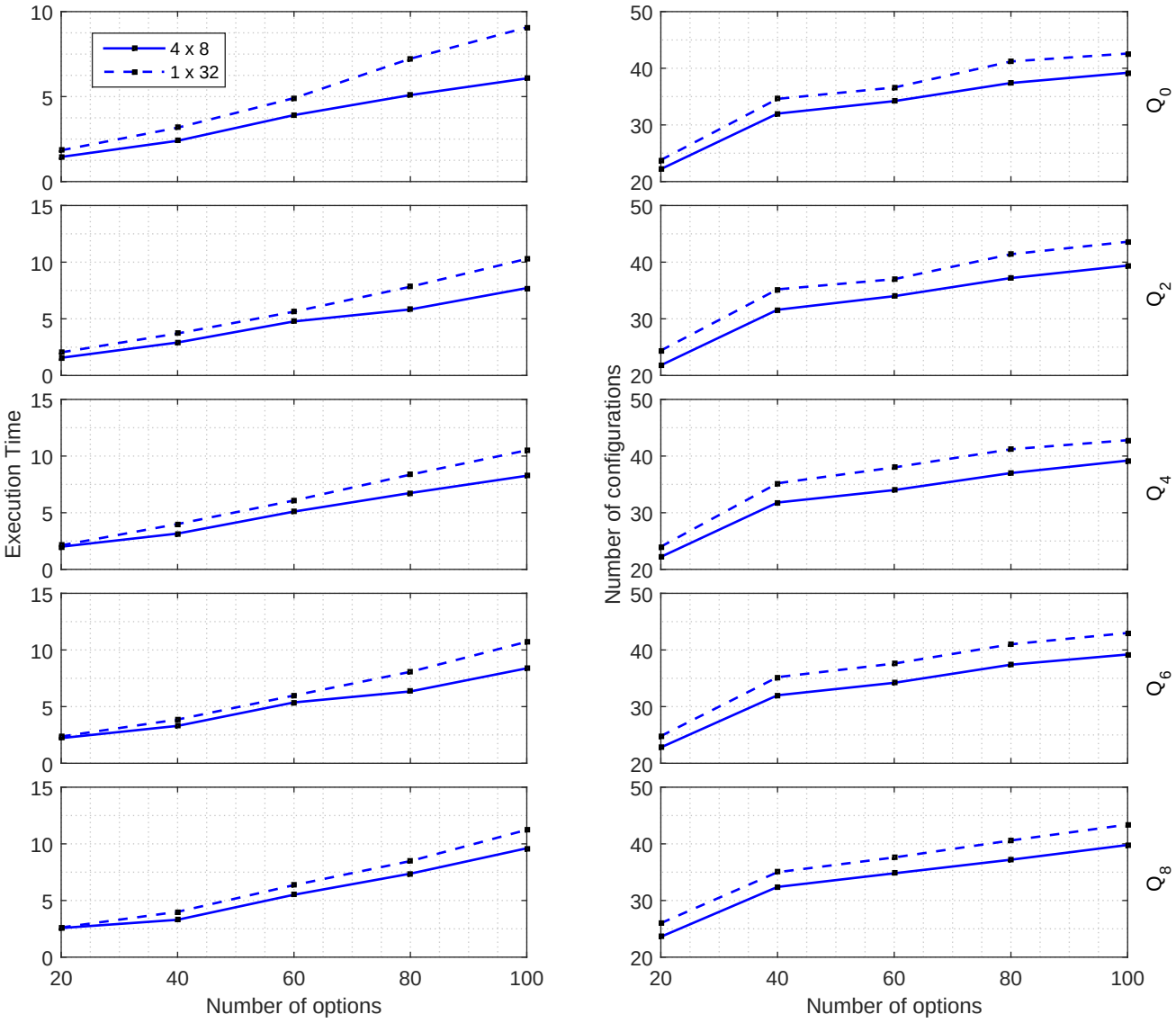

Figure 5.8: Comparing execution time and size results of 1x32 and 4x8 systems for t=2 when number of constraints is fixed

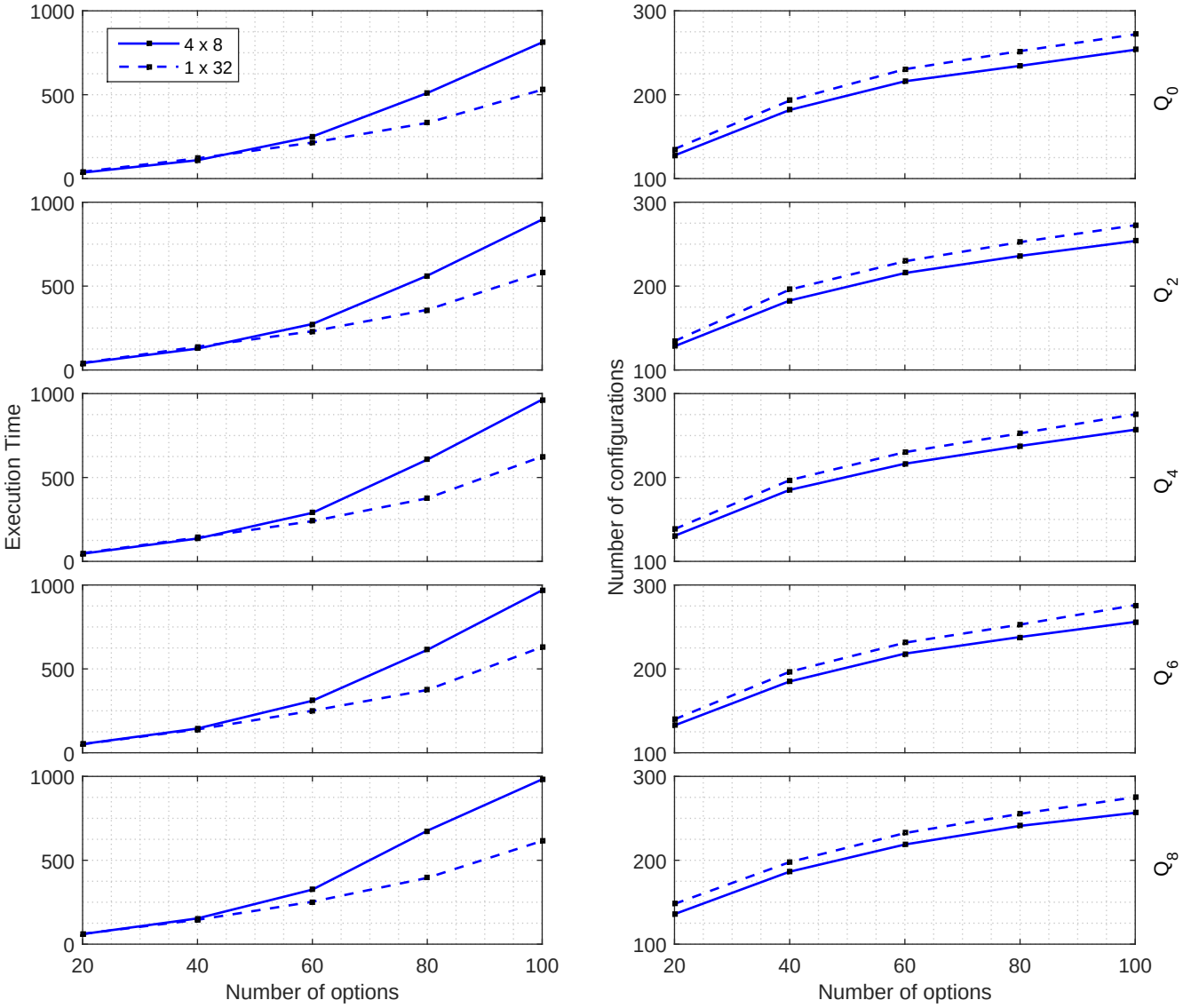

Figure 5.9: Comparing execution time and size results of 1x32 and 4x8 systems for t=3 when number of constraints is fixed

Figure 5.10 and Figure 5.11 present the results by keeping number of option fixed to observe the effect of change in number of constraints. However, for both t=2 and t=3, there are no clear dominance again, when the number of options is fixed.

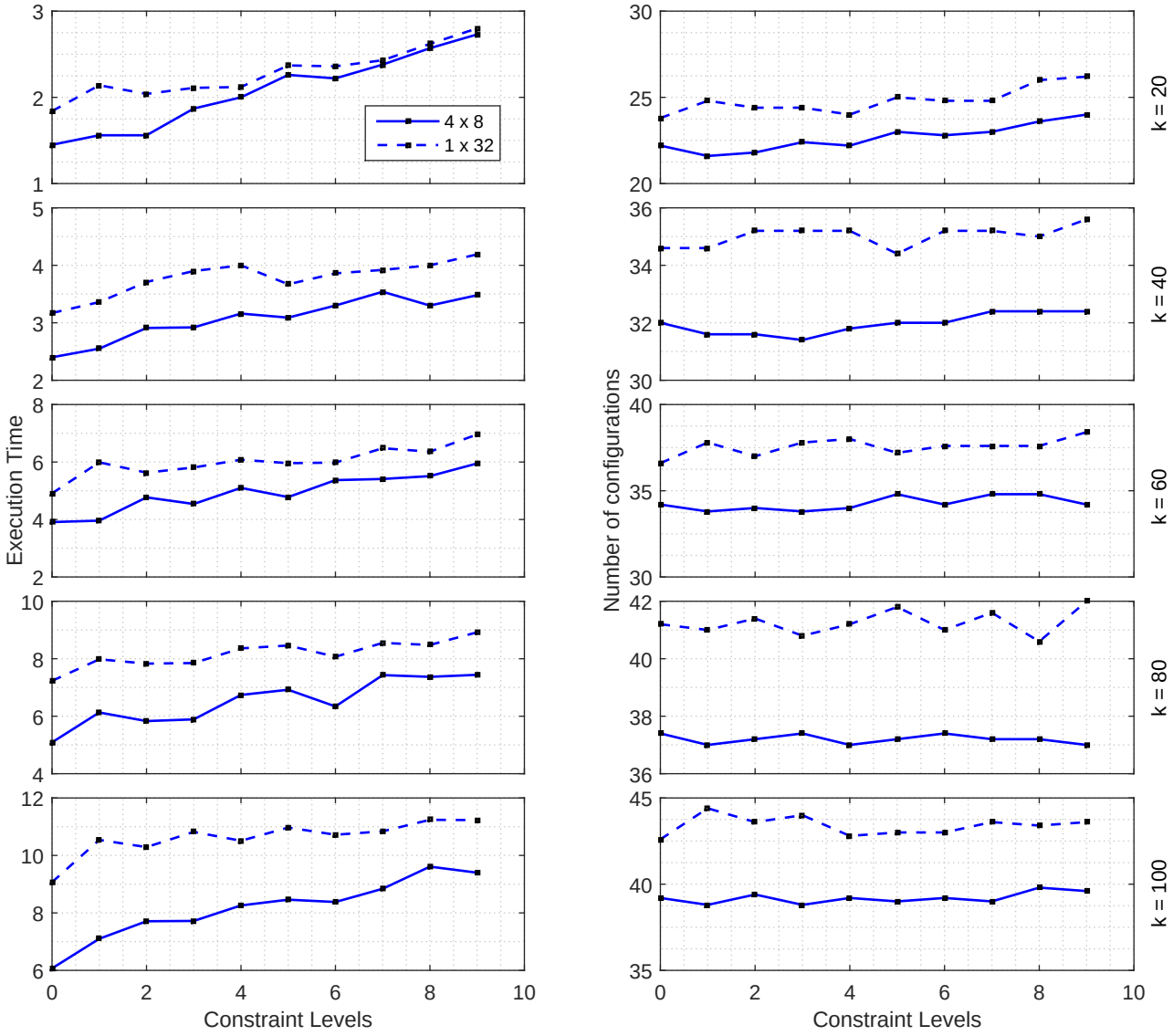

Figure 5.10: Comparing execution time and size results of 1x32 and 4x8 systems for t=2 when number of options is fixed

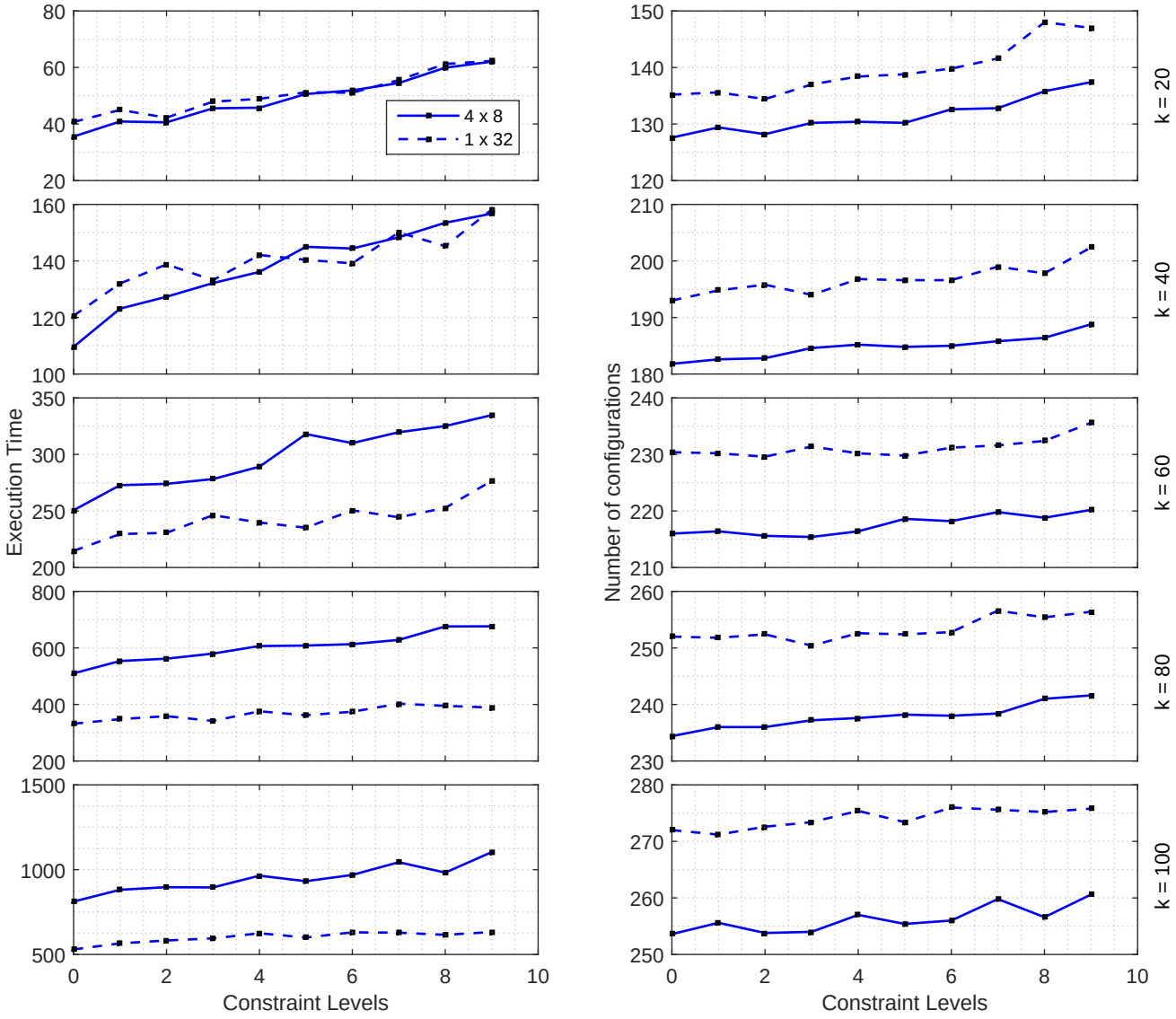

Figure 5.11: Comparing execution time and size results of 1x32 and 4x8 systems for t=3 when number of options is fixed

5.3.4. Discussions

In these experiments, we observed that multiple NSs generation algorithm constructed smaller size CAs than single NS approach for both t=2 and t=3. Additionally, multiple NSs technique constructed CAs in a shorter time when t=2 but not when t=3. We believe that this happened because of that single NS algorithm used 32 blocks and the computations were carried out much faster. On the other hand, multiple NSs generation algorithm used 8 blocks for each gain calculation of NS which increased the computation time.

In multiple NSs strategy, more than one NS were generated in each iteration and choosing the best one assisted us to converge the solution faster if it exists with the given size. Moreover, generating multiple NSs helped us to cover the corner cases of t-tuples faster which were not easy to cover due to the dense constraint distribution. Therefore, the idea of multiple NSs technique increased the quality of accepted NS in each iteration and decreased the negative effect of pure random approach. We believe that these were the reasons that multiple NSs approach constructed smaller size CAs compared to single NS generation algorithm.

Since both algorithms are better in some cases, no algorithm is superior over another one. Hence, we leave the choice of deciding number of NSs to the user.

5.4. Experiments on Hybrid Approach

Finally, in this section we present the experimental results of our final proposed approach, called hybrid algorithm (Section 4.7). In the previous part of experiments (Section 5.3), results for multiple NSs generation algorithm were already given. We used these same results to compare with hybrid approach. In the experiments, both algorithms tried to construct a complete CA with the minimal size and all procedures are included in the execution time. Hybrid approach also used 4 SAs with 8 blocks each.

5.4.1. Experimental setup

In this set of experimentations, we conducted our experiments on the same configuration space with previous part of experiments (Section 5.3). There are 2000 experiments for this comparison but only hybrid part (1000 experiments) were performed in this section.

$$\{2 \times algos\} \times \{2 \times t\} \times \{5 \times k\} \times \{10 \times Q_i\} \times \{5 \times conf.\ space\} = 1000\ experiments$$

5.4.2. Evaluation framework

We believe that hybrid approach decreases the size of CA and the execution time further compared to multiple NSs generation algorithm. Therefore, we evaluated our proposed approach both in time and size.

5.4.3. Results and analysis

Figure 5.12.(a) and Figure 5.12.(b) illustrates the number of configuration results for t=2 and t=3. When t=2, no algorithm can dominate each other i.e., they both construct smaller size CAs in some of the experiments. However, when t=3, hybrid approach mostly constructs smaller size CAs than multiple NSs generation algorithm especially as the size grows.

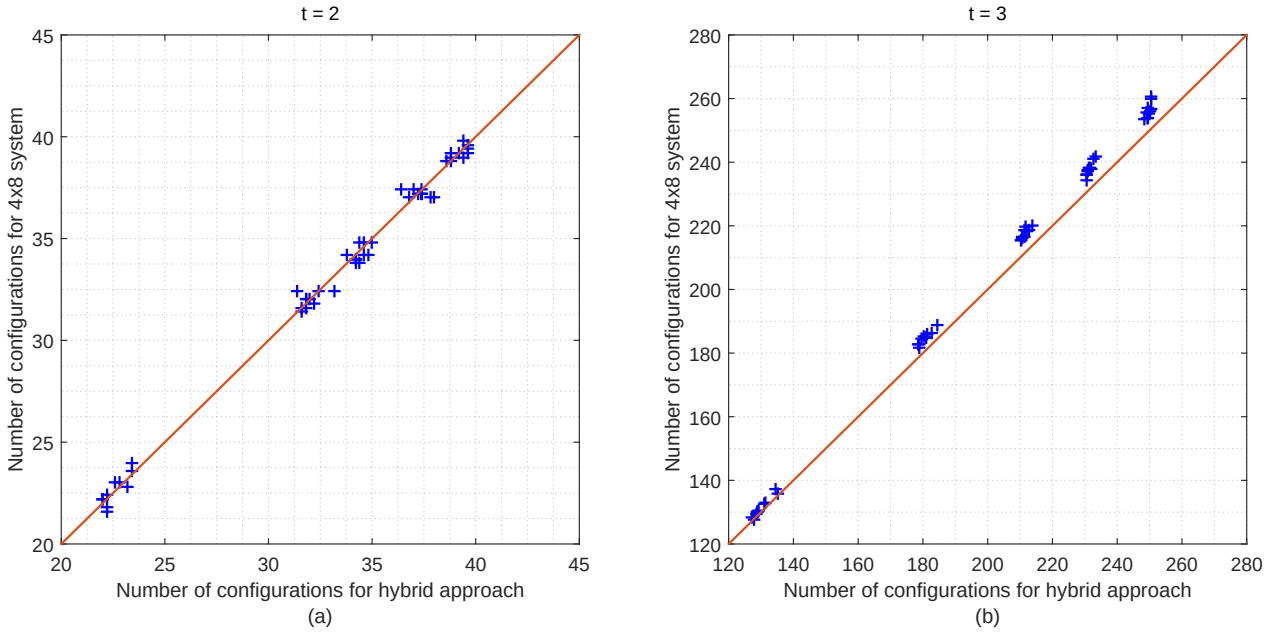

Figure 5.12: Comparing size results of multiple NSs generation and hybrid approach for t=2 and t=3

Figure 5.13.(a) and Figure 5.13.(b) illustrates the execution time results for t=2 and t=3. In these results, it is more clear that hybrid approach performs better in time for both strength values.

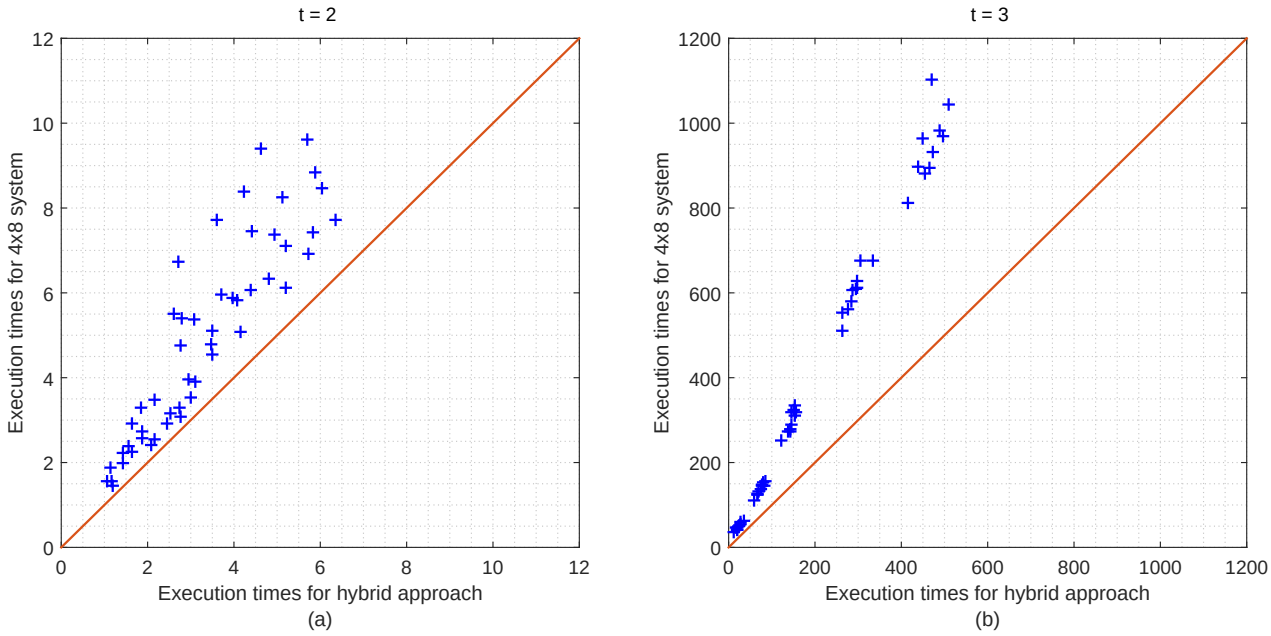

Figure 5.13: Comparing execution time results of multiple NSs generation and hybrid approach for t=2 and t=3

k	t	$\|Q_i\|$	Time		Size	
			hybrid	4x8	hybrid	4x8
20	2	0	1.20*	1.45	22*	22.2
20	2	8	1.05*	1.56	22.2	21.6*
20	2	17	1.17*	1.56	22.2	21.8*
20	2	26	1.15*	1.87	22.2*	22.4
20	2	35	1.44*	2.00	22*	22.2
20	2	44	1.65*	2.26	22.6*	23
20	2	53	1.43*	2.22	23.2	22.8*
20	2	62	1.56*	2.38	22.8*	23
20	2	71	1.87*	2.57	23.4	23.6*
20	2	80	1.87*	2.73	23.4*	24
40	2	0	2.07*	2.40	31.8*	32
40	2	12	2.17*	2.55	31.8	31.6*
40	2	25	1.64*	2.91	31.6*	31.6*
40	2	37	2.46*	2.92	31.6	31.4*
40	2	50	2.52*	3.16	32.2	31.8*
40	2	63	2.77*	3.09	32*	32*
40	2	75	2.74*	3.30	31.8*	32
40	2	88	3.00*	3.54	31.4*	32.4
40	2	101	1.85*	3.30	33.2	32.4*
40	2	113	2.17*	3.48	32.4*	32.4*
60	2	0	3.11*	3.91	33.8*	34.2
60	2	15	2.95*	3.96	34.2	33.8*
60	2	30	2.77*	4.77	34.2	34*
60	2	46	3.49*	4.54	34.4	33.8*
60	2	61	3.49*	5.10	34.2	34*
60	2	77	3.48*	4.78	34.4*	34.8
60	2	92	3.08*	5.37	34.6	34.2*
60	2	108	2.80*	5.41	34.6*	34.8
60	2	123	2.62*	5.51	35	34.8*
60	2	139	3.71*	5.95	34.8	34.2*
80	2	0	4.16*	5.09	36.4*	37.4
80	2	17	5.21*	6.13	36.8*	37
80	2	35	4.08*	5.83	37.4	37.2*
80	2	53	3.98*	5.89	37.4*	37.4*
80	2	71	2.72*	6.74	38	37*
80	2	89	5.73*	6.92	37.4	37.2*
80	2	107	4.81*	6.34	37*	37.4
80	2	125	5.84*	7.43	37.2*	37.2*
80	2	143	4.95*	7.37	37.4	37.2*
80	2	160	4.41*	7.44	37.8	37*
100	2	0	4.38*	6.07	38.8*	39.2
100	2	20	5.20*	7.10	38.8*	38.8*
100	2	40	3.60*	7.71	39.6	39.4*
100	2	60	6.35*	7.72	38.6*	38.8
100	2	80	5.11*	8.26	39.2*	39.2*
100	2	100	6.04*	8.46	38.8*	39
100	2	120	4.22*	8.38	39.6	39.2*
100	2	140	5.88*	8.84	39.4	39*
100	2	160	5.71*	9.61	39.4*	39.8
100	2	180	4.62*	9.40	39.6*	39.6*

(a)

k	t	$\|Q_i\|$	Time		Size	
			hybrid	4x8	hybrid	4x8
20	3	0	12.32*	35.56	127.8	127.6*
20	3	8	18.79*	40.89	128.4*	129.4
20	3	17	19.66*	40.59	127.2*	128.2
20	3	26	18.84*	45.56	128.8*	130.2
20	3	35	17.87*	45.76	129.4*	130.4
20	3	44	21.49*	50.61	129*	130.2
20	3	53	25.56*	51.90	131*	132.6
20	3	62	27.58*	54.50	131.4*	132.8
20	3	71	28.45*	59.92	135.2*	135.8
20	3	80	34.99*	62.07	134.6*	137.4
40	3	0	59.93*	109.63	178.8*	181.8
40	3	12	65.70*	123.13	178.6*	182.6
40	3	25	67.39*	127.34	179*	182.8
40	3	37	69.58*	132.18	179.4*	184.6
40	3	50	75.81*	136.11	180.4*	185.2
40	3	63	77.58*	144.99	181*	184.8
40	3	75	83.69*	144.41	181*	185
40	3	88	77.07*	148.37	181.4*	185.8
40	3	101	79.40*	153.51	182.6*	186.4
40	3	113	84.22*	156.74	184.6*	188.8
60	3	0	122.46*	250.69	210.6*	216
60	3	15	136.84*	272.81	210.8*	216.4
60	3	30	141.54*	273.99	210.2*	215.6
60	3	46	143.95*	278.22	210.4*	215.4
60	3	61	146.05*	289.14	211.2*	216.4
60	3	77	156.14*	318.01	211.4*	218.6
60	3	92	153.03*	310.02	212*	218.2
60	3	108	145.98*	319.65	211.8*	219.8
60	3	123	149.83*	325.13	212.6*	218.8
60	3	139	153.96*	334.75	213.8*	220.2
80	3	0	263.48*	510.08	230.4*	234.4
80	3	17	262.03*	553.74	230.6*	236
80	3	35	277.48*	561.91	230.6*	236
80	3	53	283.10*	579.98	231*	237.2
80	3	71	287.71*	606.66	231*	237.6
80	3	89	293.64*	608.38	231.2*	238.2
80	3	107	296.37*	613.21	232*	238
80	3	125	297.91*	628.40	231.6*	238.4
80	3	143	305.37*	675.96	232.6*	241
80	3	160	333.76*	676.37	233.2*	241.6
100	3	0	414.07*	811.97	248.4*	253.6
100	3	20	454.67*	881.37	249.2*	255.6
100	3	40	439.18*	896.77	249.6*	253.8
100	3	60	463.84*	895.21	249.6*	254
100	3	80	449.41*	964.56	249.6*	257
100	3	100	472.82*	931.90	249.8*	255.4
100	3	120	497.09*	968.98	250.2*	256
100	3	140	509.03*	1043.84	250.4*	259.8
100	3	160	489.03*	982.54	250.6*	256.6
100	3	180	471.49*	1103.46	250.6*	260.6

(b)

Figure 5.14: Comparison of hybrid approach and multiple NSs generation (4x8) algorithm (a) for t=2 and (b) t=3

Figure 5.14 presents all the results for multiple NSs generation and hybrid approach in separate tables for t=2 and t=3. When we analyse the execution times for both t=2 and t=3 results, hybrid approach is always better as we already give in Figure 5.13. On the other hand, when t=2, sizes of CAs for both algorithms almost equal to each other. No algorithm can obviously surpass the other one. However, when t=3, hybrid approach constructs CAs in smaller size except 1 configuration space.

5.4.4. Discussions

Especially in the large configuration spaces, the inner search takes more time due to the fact that fitness function operates in a longer time and the cooling rate gets smaller. Therefore, decreasing the number of inner searches may reduce the overall execution time, significantly.

Since all we did in hybrid approach is to add a new functionality to multiple NSs generation algorithm that is to cover remaining uncovered t-tuples with SAT solver without altering any part of inner search, we can comment that SAT solver indeed decreases the number of performed inner searches. Using SAT solver clearly affects both CA sizes and construction times in most of the experiments.

5.5. Experiments on Existing Tools

In these final set of experiments, we compared our novel hybrid approach with the well known existing tools e.g., ACTS, CASA, Jenny and PICT.

5.5.1. Experimental setup

The experiments for every tool are conducted on the same configuration space with Section 5.2.1. There are total 5000 experiments for this comparisons.

$$\{5 \times tools\} \times \{2 \times t\} \times \{10 \times k\} \times \{10 \times Q_i\} \times \{5 \times conf.\ space\} = 5000\ experiments$$

5.5.2. Evaluation framework

We evaluated our algorithm in both size and time with several aspects e.g., keeping the number of constraints or options fixed.

5.5.3. Results and analysis

In this section, all results for every tool are presented in several ways to observe the best part of algorithms and compare with the hybrid approach. First of all, Figure 5.15 illustrates the execution time and size results for t=2. The execution times for algorithms are not very clear since ACTS and CASA took more time to construct CAs compared to other ones. Even though CASA is the best one in number of configurations, it is also the worst one in execution time among all tool results. Since size results of hybrid approach are very close to CASA's and the gap between execution times of algorithms reaches up to 500 seconds in some of the experiments, we remove CASA's results from the graphics. Moreover, ACTS is better than us in neither time nor size. Therefore, in order to scale the time in graphic for better visualisation, we remove ACTS results, too. New graphics are given in Figure 5.16.

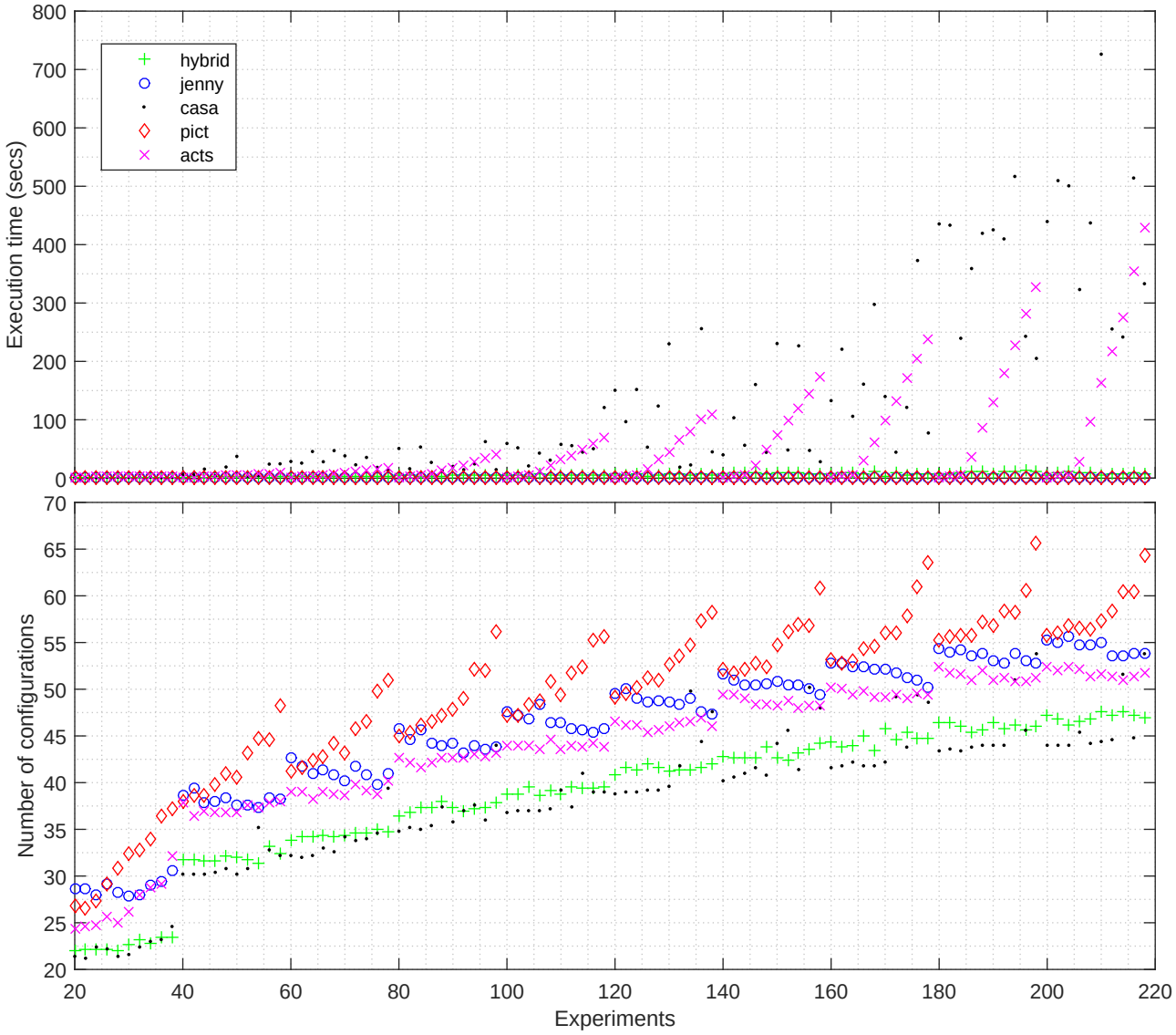

Figure 5.15: Comparing size and execution time results where t=2 for hybrid approach, Jenny, CASA, PICT and ACTS

In Figure 5.16, the execution times of Jenny and PICT are almost equal to each other and they are both under 2 seconds. However, hybrid approach reaches almost 15 seconds in some of the experiments. On the other hand, our algorithm results are better in number of configurations by 18% on average.

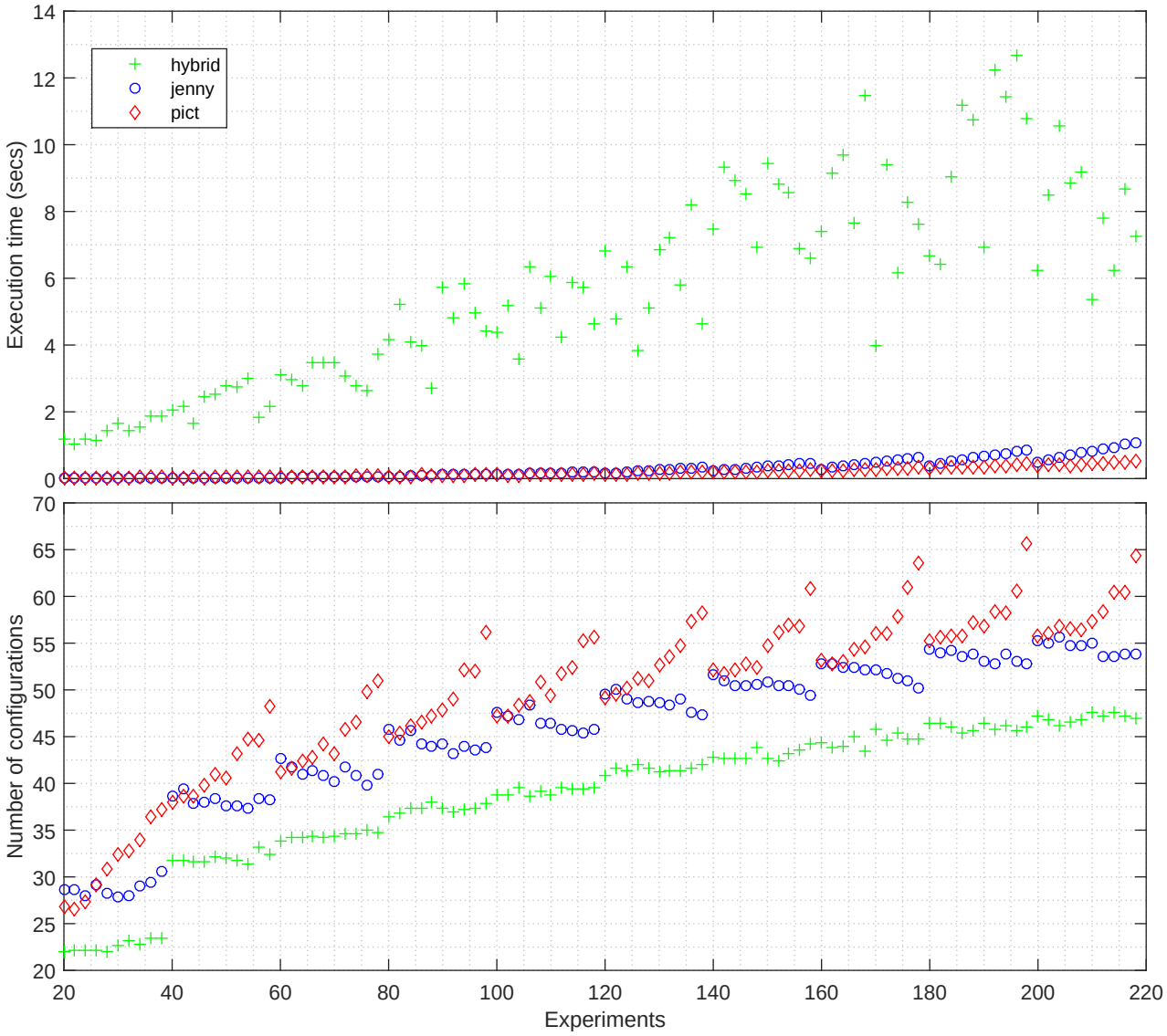

Figure 5.16: Comparing size and execution time results where t=2 for hybrid approach, Jenny and PICT

In the following figures, the results for the case of t=3 are given. We do not give CASA results for t=3 due to the time limitations.

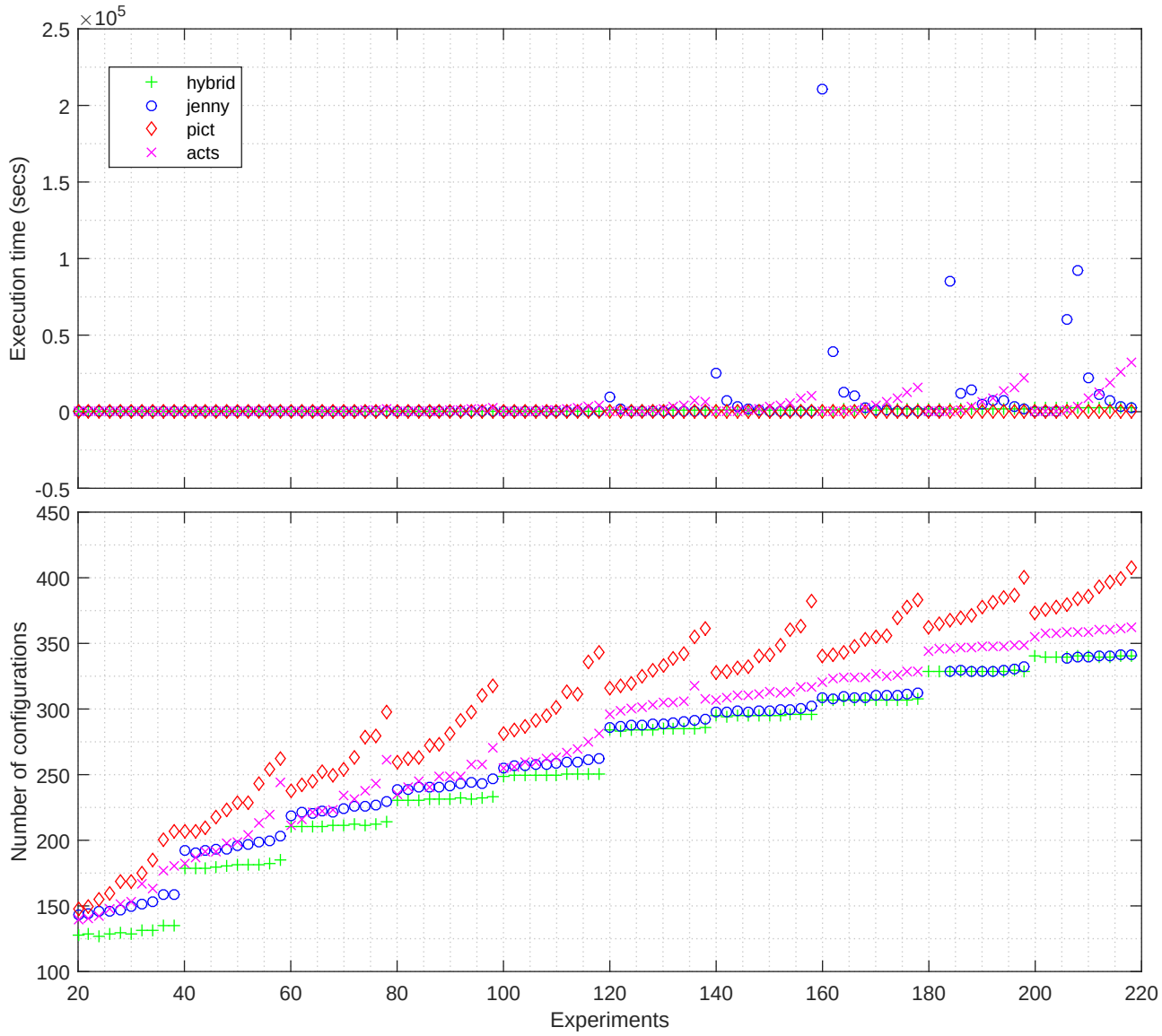

Figure 5.17: Comparing size and execution time results where t=3 for hybrid approach, Jenny, PICT and ACTS

As number of configurations increases, in general, hybrid approach is the best one among others and the closest one to our algorithm is Jenny, even sometimes better. However, in the execution time, Jenny's results is the worst one among all tools. So that we remove Jenny results form Figure 5.17 to scale the remaining tool results and observe them clearly. We illustrate the new results in Figure 5.18

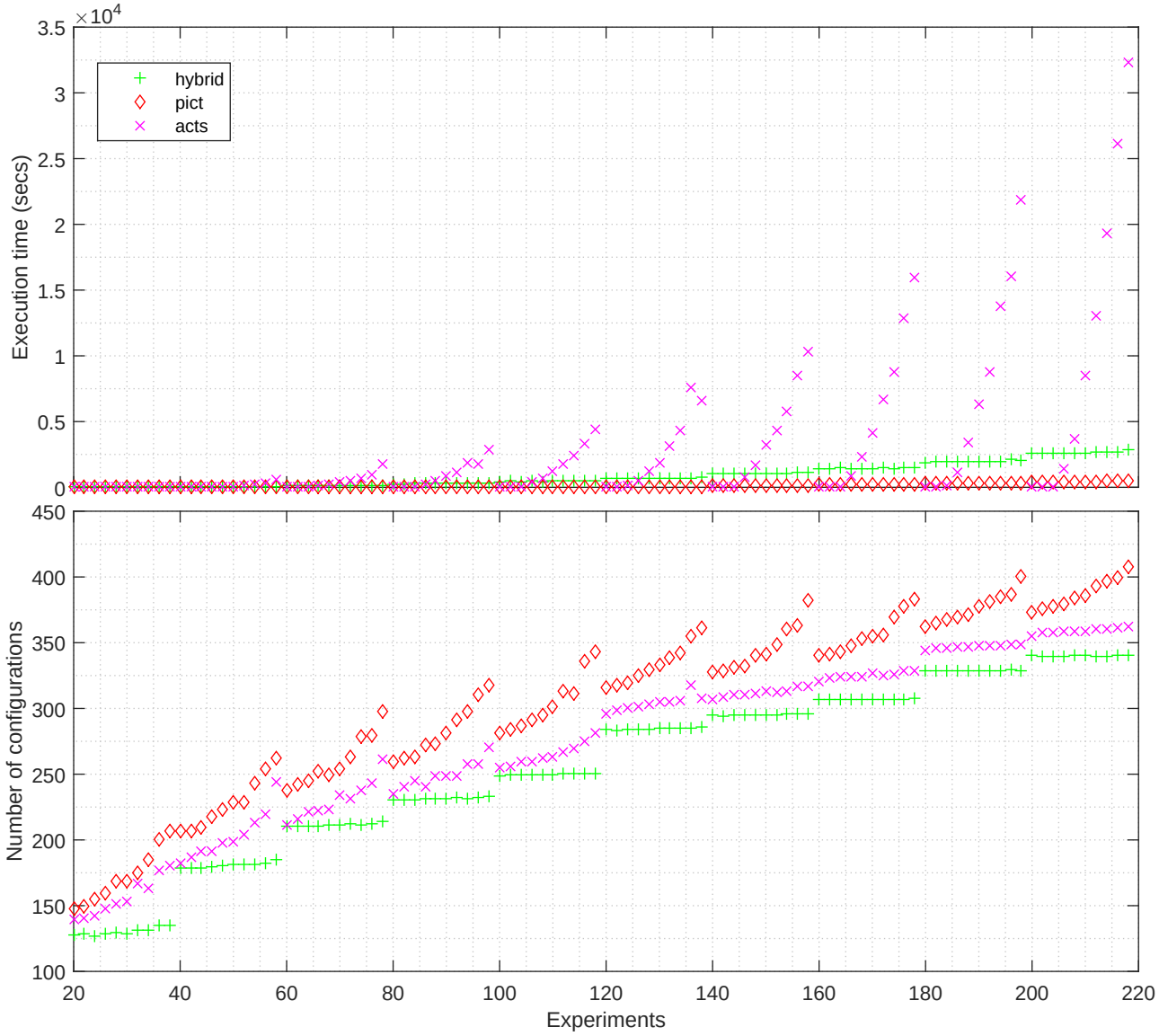

Figure 5.18: Comparing size and execution time results where t=3 for hybrid approach, PICT and ACTS

In these results, it is clear that hybrid algorithm is the best one in number of configurations however PICT results are better than our algorithm results in execution time. In order to see the relation between these two algorithms, we give results of only PICT and our algorithm with box plot graphics in Figure 5.19

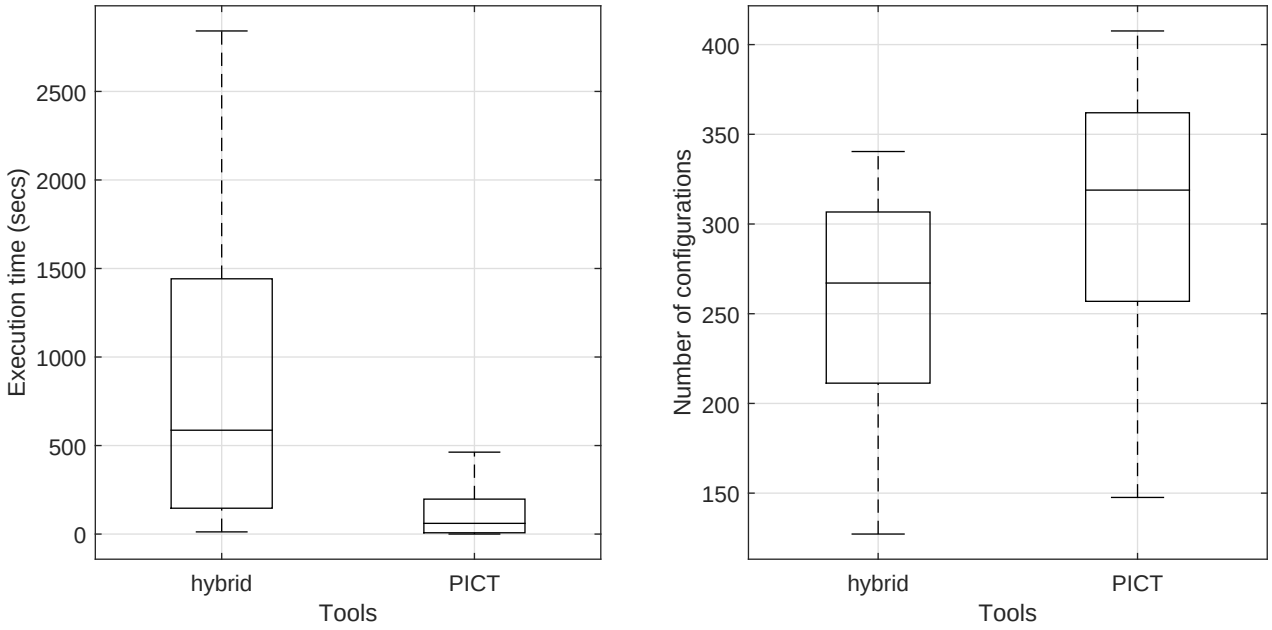

Figure 5.19: Comparing size and execution time results where t=3 for hybrid approach and PICT

Figure 5.19 indicates that both algorithms are better in one part. PICT dominates our algorithm in time, on the other hand, our algorithm constructs smaller size CAs than PICT. This conclusion is expected as we mentioned in Section 3. PICT was designed for considering three principles and one of them was constructing CAs faster. Smaller size CAs were their least priority after other three principles.

As final, we present all the experimental results of tools in Table 5.1, Table 5.2, Table 5.3 and Table 5.4.

5.5.4. Discussions

When we compare all tools in size of constructed CA, we can conclude that our proposed approach is the best one when t=3, and the second best one when t=2, in general. However, the best tool in t=2 CASA, is the worst one in execution time far from all tool results. Therefore, we believe that it is not practical to spend too much time only for t=2. As we compare our results in execution time, even though our results are not the best one, they are reasonable times compared to other tools.

Moreover, we also observe in tables that our algorithm is among the least effected ones by the number of constraints. We conjecture that this speciality of us carries big importance especially in the practical scenarios.

Based on all these observations, we conclude that our proposed algorithm is the best one as pareto optimal.

			Time					Size				
k	**t**	$\|Q_i\|$	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**
20	2	0	1.20	0.00	0.79	0.02	0.04	22.0	28.6	21.4	26.8	24.4
20	2	8	1.05	0.00	0.95	0.02	1.51	22.2	28.6	21.2	26.6	24.6
20	2	17	1.17	0.00	0.65	0.02	2.09	22.2	28.0	22.4	27.4	24.8
20	2	26	1.15	0.00	1.46	0.02	2.34	22.2	29.2	22.2	29.2	25.6
20	2	35	1.44	0.00	1.92	0.03	2.43	22.0	28.2	21.4	30.8	25.0
20	2	44	1.65	0.00	1.08	0.03	2.67	22.6	27.8	21.6	32.4	26.2
20	2	53	1.43	0.00	3.70	0.03	3.01	23.2	28.0	22.4	32.8	28.0
20	2	62	1.56	0.00	1.99	0.04	3.24	22.8	29.0	23.0	34.0	28.8
20	2	71	1.87	0.00	0.70	0.04	3.52	23.4	29.4	23.2	36.4	29.2
20	2	80	1.87	0.00	1.50	0.04	4.06	23.4	30.6	24.6	37.2	32.2
40	2	0	2.07	0.01	5.75	0.03	0.07	31.8	38.6	30.2	38.0	37.8
40	2	12	2.17	0.02	5.55	0.03	2.06	31.8	39.4	30.2	38.6	36.4
40	2	25	1.64	0.02	15.52	0.04	3.14	31.6	37.8	30.2	38.6	37.0
40	2	37	2.46	0.02	3.86	0.03	3.95	31.6	38.0	30.4	39.8	36.8
40	2	50	2.52	0.02	19.11	0.04	4.69	32.2	38.4	30.8	41.0	36.8
40	2	63	2.77	0.02	37.33	0.04	5.46	32.0	37.6	30.2	40.6	36.8
40	2	75	2.74	0.02	3.40	0.05	6.21	31.8	37.6	30.8	43.2	37.6
40	2	88	3.00	0.02	4.49	0.06	6.94	31.4	37.4	35.2	44.8	37.4
40	2	101	1.85	0.03	24.04	0.06	7.96	33.2	38.4	32.8	44.6	37.8
40	2	113	2.17	0.03	24.56	0.06	8.78	32.4	38.2	32.2	48.2	38.0
60	2	0	3.11	0.03	28.62	0.05	0.08	33.8	42.6	32.2	41.2	39.0
60	2	15	2.95	0.04	25.79	0.05	2.12	34.2	41.8	32.0	41.6	39.0
60	2	30	2.77	0.05	45.64	0.06	3.64	34.2	41.0	32.2	42.4	38.2
60	2	46	3.49	0.05	28.10	0.05	5.50	34.4	41.4	33.0	42.8	39.0
60	2	61	3.49	0.05	47.17	0.06	7.69	34.2	40.8	32.6	44.2	38.8
60	2	77	3.48	0.06	38.22	0.06	9.25	34.4	40.2	34.2	43.2	38.6
60	2	92	3.08	0.06	22.86	0.06	11.14	34.6	41.8	33.8	45.8	39.8
60	2	108	2.80	0.06	35.50	0.08	12.74	34.6	40.8	34.0	46.6	39.2
60	2	123	2.62	0.07	18.41	0.08	14.61	35.0	39.8	34.6	49.8	38.8
60	2	139	3.71	0.07	13.62	0.09	17.92	34.8	41.0	39.4	51.0	40.2
80	2	0	4.16	0.07	50.84	0.07	0.10	36.4	45.8	34.8	45.0	42.6
80	2	17	5.21	0.07	16.14	0.07	2.24	36.8	44.6	35.2	45.4	42.2
80	2	35	4.08	0.09	53.34	0.07	4.58	37.4	45.6	35.0	46.2	41.6
80	2	53	3.98	0.10	26.96	0.14	7.67	37.4	44.2	35.4	46.6	42.2
80	2	71	2.72	0.10	13.06	0.09	12.60	38.0	44.0	37.4	47.2	42.6
80	2	89	5.73	0.11	20.15	0.09	17.47	37.4	44.2	35.8	47.8	42.6
80	2	107	4.81	0.12	14.50	0.09	21.78	37.0	43.2	37.0	49.0	42.6
80	2	125	5.84	0.12	24.41	0.10	28.17	37.2	44.0	37.6	52.2	43.0
80	2	143	4.95	0.13	62.56	0.11	33.56	37.4	43.6	36.0	52.0	42.8
80	2	160	4.41	0.13	14.58	0.12	39.55	37.8	43.8	44.0	56.2	43.2
100	2	0	4.38	0.11	59.41	0.11	0.11	38.8	47.6	36.8	47.2	44.0
100	2	20	5.20	0.12	51.99	0.10	2.26	38.8	47.2	37.0	47.2	44.0
100	2	40	3.60	0.14	20.96	0.10	5.12	39.6	46.8	37.0	48.4	44.0
100	2	60	6.35	0.15	42.86	0.11	11.92	38.6	48.4	37.0	48.8	43.6
100	2	80	5.11	0.16	30.84	0.11	20.88	39.2	46.4	37.2	50.8	44.6
100	2	100	6.04	0.18	57.84	0.12	32.14	38.8	46.4	39.2	49.4	43.6
100	2	120	4.22	0.18	55.67	0.12	39.17	39.6	45.8	37.4	51.8	44.0
100	2	140	5.88	0.19	44.62	0.13	48.53	39.4	45.6	41.0	52.4	43.8
100	2	160	5.71	0.21	50.43	0.14	58.86	39.4	45.4	39.0	55.2	44.2
100	2	180	4.62	0.21	121.01	0.16	68.77	39.6	45.8	39.0	55.6	43.8

Table 5.1: Experimental results for all tools where t=2 and k $\in \{20, 40, 60, 80, 100\}$

			Time					Size				
k	**t**	$\|Q_i\|$	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**
120	2	0	6,80	0,16	150,53	0,14	0,13	40,8	49,6	38,8	49,2	46,6
120	2	21	4,79	0,18	96,72	0,14	2,42	41,6	50,0	39,0	49,6	46,2
120	2	43	6,35	0,20	151,89	0,14	5,41	41,4	49,0	39,0	50,2	46,2
120	2	65	3,82	0,23	53,06	0,15	14,77	42,0	48,6	39,2	51,2	45,4
120	2	87	5,12	0,25	123,53	0,16	32,57	41,6	48,8	39,2	51,0	45,6
120	2	109	6,87	0,26	230,08	0,16	45,21	41,2	48,6	39,6	52,6	46,0
120	2	131	7,22	0,28	18,87	0,17	66,14	41,4	48,4	41,8	53,6	46,4
120	2	153	5,79	0,30	22,59	0,19	79,03	41,4	49,0	49,8	54,8	46,6
120	2	175	8,20	0,31	256,07	0,21	101,36	41,6	47,6	44,4	57,4	47,0
120	2	197	4,65	0,33	45,02	0,21	110,08	42,0	47,4	47,6	58,2	46,0
140	2	0	7,47	0,22	39,92	0,19	0,16	42,8	51,6	40,2	52,2	49,4
140	2	23	9,32	0,26	103,46	0,20	2,60	42,6	51,0	40,6	51,8	49,4
140	2	47	8,93	0,29	56,31	0,19	5,68	42,6	50,4	41,0	52,2	49,0
140	2	70	8,51	0,32	160,37	0,21	22,46	42,6	50,4	41,6	52,8	48,4
140	2	94	6,92	0,35	44,23	0,20	47,85	43,8	50,6	40,8	52,4	48,4
140	2	118	9,44	0,37	230,53	0,22	73,39	42,6	50,8	44,2	54,8	48,2
140	2	141	8,83	0,40	48,23	0,24	98,56	42,4	50,4	45,6	56,2	48,8
140	2	165	8,56	0,42	226,63	0,24	120,36	43,2	50,4	41,4	57,0	48,0
140	2	189	6,89	0,44	47,47	0,25	144,77	43,6	50,0	50,2	56,8	48,2
140	2	212	6,61	0,47	28,15	0,27	173,41	44,2	49,4	48,0	60,8	48,2
160	2	0	7,41	0,29	132,78	0,25	0,18	44,4	52,8	41,6	53,2	50,2
160	2	25	9,14	0,33	220,81	0,25	2,69	43,8	52,8	41,8	52,8	50,0
160	2	50	9,68	0,39	105,93	0,25	5,29	44,0	52,4	42,2	53,0	49,4
160	2	75	7,65	0,43	160,98	0,26	29,57	45,0	52,4	41,8	54,4	49,8
160	2	101	11,48	0,46	297,62	0,28	62,33	43,4	52,2	41,8	54,6	49,2
160	2	126	3,97	0,50	139,84	0,28	99,19	45,8	52,2	42,2	56,0	49,2
160	2	151	9,39	0,53	44,45	0,31	132,58	44,6	51,8	49,2	56,0	49,4
160	2	177	6,16	0,57	121,21	0,31	171,24	45,4	51,2	43,8	57,8	49,0
160	2	202	8,28	0,60	372,80	0,32	204,85	44,8	51,0	49,4	61,0	49,6
160	2	227	7,63	0,63	77,38	0,33	238,53	44,8	50,2	48,6	63,6	49,4
180	2	0	6,68	0,39	435,49	0,32	0,20	46,4	54,4	43,4	55,2	52,4
180	2	26	6,42	0,45	433,30	0,33	2,79	46,4	54,0	43,6	55,6	51,8
180	2	53	9,03	0,51	239,58	0,34	5,22	46,0	54,2	43,4	55,8	51,6
180	2	80	11,18	0,57	359,03	0,34	35,83	45,4	53,6	43,8	55,8	51,0
180	2	107	10,75	0,63	419,38	0,36	86,28	45,6	53,8	44,0	57,2	52,0
180	2	134	6,93	0,67	425,39	0,36	129,29	46,4	53,0	44,0	56,8	51,0
180	2	160	12,23	0,71	409,82	0,39	180,66	45,8	52,8	44,0	58,4	51,2
180	2	187	11,43	0,76	516,80	0,39	226,60	46,2	53,8	51,0	58,2	50,8
180	2	214	12,66	0,81	242,98	0,41	281,84	45,6	53,0	45,6	60,6	50,8
180	2	241	10,77	0,85	205,12	0,44	327,40	46,0	52,8	53,8	65,6	51,2
200	2	0	6,23	0,48	439,55	0,40	0,24	47,2	55,2	44,0	55,8	52,4
200	2	28	8,48	0,56	509,66	0,41	2,58	46,8	55,0	44,0	56,0	52,0
200	2	56	10,55	0,63	500,69	0,41	5,00	46,2	55,6	44,0	56,8	52,4
200	2	84	8,85	0,71	323,07	0,40	29,09	46,6	54,8	45,4	56,6	52,2
200	2	113	9,20	0,78	437,17	0,42	96,28	46,8	54,8	44,2	56,4	51,4
200	2	141	5,35	0,82	726,25	0,45	162,84	47,6	55,0	44,4	57,4	51,6
200	2	169	7,81	0,88	255,51	0,46	217,09	47,2	53,6	44,6	58,4	51,4
200	2	197	6,24	0,93	241,75	0,48	274,80	47,6	53,6	51,6	60,4	51,0
200	2	226	8,68	1,03	514,13	0,50	354,82	47,2	53,8	44,8	60,4	51,4
200	2	254	7,26	1,07	333,06	0,52	429,18	47,0	53,8	53,8	64,4	51,8

Table 5.2: Experimental results for all tools where t=2 and k $\in \{120, 140, 160, 180, 200\}$

			Time					Size				
k	**t**	$\|Q_i\|$	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**
20	3	0	12.32	0.17	2900.65	0.15	0.08	127.8	143.2	124.4	147.6	139.8
20	3	8	18.79	0.16	1214.09	9.22	2.57	128.4	144.2	126.0	150.0	140.2
20	3	17	19.66	0.15	49.90	0.16	4.72	127.2	145.6	129.6	155.4	142.0
20	3	26	18.84	0.15	277.57	0.16	6.86	128.8	146.0	127.0	159.4	147.4
20	3	35	17.87	0.15	125.28	0.17	9.10	129.4	147.2	129.0	169.0	151.0
20	3	44	21.49	0.15	195.81	0.17	11.51	129.0	149.4	131.0	168.6	153.2
20	3	53	25.56	0.15	1243.29	0.18	17.72	131.0	151.6	132.0	175.0	166.4
20	3	62	27.58	0.15	174.38	0.18	20.98	131.4	153.4	134.2	185.4	162.8
20	3	71	28.45	0.16	452.88	0.20	33.01	135.2	158.2	136.2	200.4	177.2
20	3	80	34.99	0.15	39.95	0.21	49.04	134.6	159.0	139.2	207.0	180.6
40	3	0	59.93	2.50	21514.33	1.61	0.22	178.8	192.0	169.0	206.8	182.0
40	3	12	65.70	2.32	-	9.01	3.55	178.6	190.6	-	207.2	186.4
40	3	25	67.39	2.17	6896.67	1.65	13.28	179.0	192.2	178.5	209.4	191.0
40	3	37	69.58	2.09	1356.80	1.58	29.08	179.4	193.0	183.2	217.4	191.4
40	3	50	75.81	2.02	1801.15	1.67	63.30	180.4	193.6	182.6	223.6	197.4
40	3	63	77.58	1.94	2541.76	1.64	89.14	181.0	196.0	187.0	228.8	198.4
40	3	75	83.69	1.97	3056.12	1.76	129.80	181.0	197.2	185.0	229.0	204.2
40	3	88	77.07	1.93	484.79	1.79	230.93	181.4	199.0	188.6	242.8	213.4
40	3	101	79.40	1.89	1513.62	1.91	295.08	182.6	200.0	188.2	254.4	219.2
40	3	113	84.22	1.94	637.01	1.97	595.55	184.6	203.4	193.6	262.6	243.8
60	3	0	122.46	14.18	-	6.64	0.50	210.6	219.0	-	237.8	211.2
60	3	15	136.84	12.33	-	15.38	4.27	210.8	221.2	-	242.2	216.2
60	3	30	141.54	10.09	-	6.69	29.66	210.2	220.8	-	244.8	221.2
60	3	46	143.95	9.38	-	6.87	125.85	210.4	222.0	-	252.2	222.2
60	3	61	146.05	9.26	-	6.70	193.51	211.2	221.6	-	249.2	223.4
60	3	77	156.14	8.65	-	6.74	384.78	211.4	224.2	-	254.0	234.4
60	3	92	153.03	8.21	19496.63	6.95	496.70	212.0	225.6	217.6	263.4	231.0
60	3	108	145.98	7.97	3655.34	7.33	681.72	211.8	226.2	224.2	278.6	237.8
60	3	123	149.83	7.79	4219.94	7.33	924.26	212.6	226.8	221.8	279.6	243.2
60	3	139	153.96	7.91	17344.14	7.68	1729.18	213.8	229.4	223.2	298.0	261.4
80	3	0	263.48	55.45	-	16.14	0.97	230.4	238.6	-	259.4	235.0
80	3	17	262.03	45.37	-	25.05	4.98	230.6	238.6	-	262.2	240.8
80	3	35	277.48	29.84	-	16.12	49.01	230.6	240.4	-	263.4	245.4
80	3	53	283.10	25.23	-	16.60	262.34	231.0	240.4	-	272.6	240.4
80	3	71	287.71	21.83	-	16.47	505.27	231.0	240.8	-	273.0	248.4
80	3	89	293.64	20.27	-	17.02	857.00	231.2	241.8	-	281.4	249.0
80	3	107	296.37	18.84	-	17.31	1103.06	232.0	243.0	-	291.0	248.4
80	3	125	297.91	18.96	-	17.68	1888.72	231.6	244.4	-	297.4	257.4
80	3	143	305.37	18.68	-	18.25	1808.36	232.6	243.2	-	310.4	257.6
80	3	160	333.76	18.38	-	18.55	2903.37	233.2	246.8	-	317.6	270.4
100	3	0	414.07	634.02	-	36.13	1.96	248.4	255.2	-	281.2	255.2
100	3	20	454.67	226.42	-	45.29	5.79	249.2	256.6	-	283.8	256.0
100	3	40	439.18	176.22	-	36.28	70.75	249.6	257.0	-	286.4	259.6
100	3	60	463.84	81.49	-	36.50	379.90	249.6	257.4	-	291.8	259.2
100	3	80	449.41	63.55	-	36.95	708.01	249.6	257.8	-	295.0	262.0
100	3	100	472.82	52.22	-	37.74	1242.08	249.8	258.4	-	301.0	263.6
100	3	120	497.09	47.77	-	38.58	1812.76	250.2	259.8	-	313.4	266.4
100	3	140	509.03	46.11	-	38.63	2375.59	250.4	260.0	-	311.4	269.8
100	3	160	489.03	43.36	-	40.87	3282.93	250.6	261.4	-	335.6	274.8
100	3	180	471.49	41.69	-	41.40	4394.43	250.6	262.2	-	342.8	281.6

Table 5.3: Experimental results for all tools where t=3 and k $\in \{20, 40, 60, 80, 100\}$

			Time					Size				
k	**t**	$\|Q_i\|$	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**	**hybrid**	**Jenny**	**CASA**	**PICT**	**ACTS**
120	3	0	664.12	9947.21	-	76.02	3.74	284.2	286.2	-	316.2	296.2
120	3	21	681.31	2005.56	-	80.33	9.29	283.6	287.2	-	318.0	298.4
120	3	43	690.76	496.70	-	75.72	92.56	284.2	287.8	-	319.8	300.0
120	3	65	691.10	260.58	-	76.24	541.00	284.4	288.0	-	324.8	301.2
120	3	87	701.37	212.23	-	77.05	1245.27	284.2	289.0	-	329.6	303.0
120	3	109	704.82	150.25	-	78.47	1890.28	284.8	289.0	-	333.6	304.8
120	3	131	711.48	138.89	-	79.04	3138.15	285.2	289.6	-	339.0	304.6
120	3	153	719.33	121.30	-	80.12	4290.01	285.0	290.0	-	342.4	305.8
120	3	175	726.14	114.12	-	81.09	7551.46	285.4	291.2	-	355.2	317.6
120	3	197	736.71	104.47	-	82.40	6604.20	285.6	292.6	-	361.4	307.4
140	3	0	1015.31	25013.03	-	126.03	6.02	294.8	297.4	-	327.8	306.8
140	3	23	1036.67	7407.34	-	125.45	11.93	294.4	298.0	-	329.0	309.0
140	3	47	1049.68	3493.25	-	125.00	70.45	295.2	298.4	-	331.2	310.6
140	3	70	1051.53	1746.47	-	126.17	806.37	295.2	297.6	-	332.4	310.2
140	3	94	1063.95	661.84	-	127.48	1705.30	294.8	298.4	-	340.6	311.2
140	3	118	1066.42	376.80	-	129.07	3259.33	295.0	298.2	-	341.8	312.8
140	3	141	1069.43	328.65	-	132.74	4282.14	295.4	299.2	-	348.4	312.2
140	3	165	1088.30	234.72	-	133.10	5762.59	295.6	299.8	-	360.0	313.0
140	3	189	1094.67	246.65	-	134.53	8496.18	295.8	300.8	-	362.8	316.8
140	3	212	1097.54	205.06	-	139.92	10282.93	296.2	302.0	-	382.4	316.4
160	3	0	1379.81	210370.80	-	191.50	8.87	306.6	309.0	-	340.4	320.2
160	3	25	1402.54	39526.43	-	192.05	15.85	306.6	308.0	-	341.2	323.2
160	3	50	1474.16	12367.26	-	191.66	80.77	307.0	309.4	-	342.8	324.0
160	3	75	1409.58	10621.97	-	194.05	904.70	306.4	309.0	-	348.0	324.4
160	3	101	1429.96	2482.68	-	196.73	2284.33	307.0	309.0	-	353.2	324.0
160	3	126	1427.07	1198.34	-	198.05	4115.47	306.6	310.8	-	354.8	326.4
160	3	151	1514.31	879.03	-	198.48	6647.35	306.8	310.0	-	356.2	325.4
160	3	177	1453.60	535.39	-	202.28	8770.18	306.8	310.6	-	369.4	325.6
160	3	202	1469.60	376.14	-	204.86	12867.99	306.6	311.4	-	378.0	329.0
160	3	227	1471.38	359.13	-	209.02	15995.89	307.4	312.6	-	383.2	328.4
180	3	0	1908.00	-	-	320.65	14.59	328.4	-	-	362.6	344.0
180	3	26	1931.19	-	-	308.40	23.08	328.4	-	-	365.0	345.6
180	3	53	1932.54	85186.06	-	311.68	99.75	328.6	328.6	-	367.4	346.0
180	3	80	1943.12	11731.54	-	312.00	1157.33	328.8	329.6	-	369.6	346.6
180	3	107	1957.20	14431.12	-	315.43	3389.61	328.6	328.2	-	371.0	346.8
180	3	134	1967.65	5319.23	-	319.60	6331.94	329.0	328.8	-	377.6	347.4
180	3	160	1974.40	7301.05	-	317.01	8740.97	328.4	329.0	-	381.2	347.4
180	3	187	1995.25	7359.87	-	322.57	13783.6	329.0	329.6	-	385.4	348.0
180	3	214	2092.69	2992.40	-	323.14	16075.7	329.2	330.8	-	386.6	348.6
180	3	241	2017.93	2059.96	-	331.49	21906.5	329.0	332.6	-	400.8	348.2
200	3	0	2561.37	-	-	454.17	20.1	340.0	-	-	373.4	355.4
200	3	28	2598.18	-	-	444.61	29.4	339.4	-	-	376.0	357.6
200	3	56	2592.78	-	-	443.13	81.0	339.4	-	-	377.8	357.6
200	3	84	2624.31	59940.28	-	452.99	1400.5	339.8	339.0	-	379.2	358.8
200	3	113	2624.47	92246.15	-	448.71	3646.3	340.0	339.4	-	383.8	358.4
200	3	141	2625.34	22408.84	-	449.71	8541.3	340.0	339.8	-	385.6	358.4
200	3	169	2671.23	11024.89	-	448.05	13023.5	339.6	340.2	-	393.2	360.2
200	3	197	2658.30	7218.87	-	459.37	19284.8	339.6	340.8	-	397.0	360.2
200	3	226	2676.16	3444.02	-	461.06	26152.8	340.0	341.0	-	399.2	361.8
200	3	254	2841.59	2262.05	-	462.69	32295.5	340.4	341.6	-	407.6	362.0

Table 5.4: Experimental results for all tools where t=3 and k $\in \{120, 140, 160, 180, 200\}$

6

CONCLUSION AND FUTURE WORK

In this thesis, we have focused on constructing test suites generation problem for combinatorial software interaction testing. We have developed a GPU-based parallel implementation of simulated annealing, a technique to run multiple simulated annealing concurrently and a novel hybrid approach with SAT solver to construct covering arrays in the presence of constraints. In addition to these, we have described a method to measure combinatorial coverage of any given covering array, i.e., count the number of covered t-tuples by covering array.

We conducted large scale experiments including with well-known existing tools to evaluate the efficiency of our parallel algorithms and hybrid approach. Experimental results point out that our parallel implementation of simulated annealing decreases the search time significantly, especially when the configuration space is large, and our final proposed hybrid approach mostly construct smaller size covering arrays comparing to other tools in a reasonable time. Moreover, the empirical studies suggest that our hybrid algorithm is the best approach to construct covering arrays among other presented tools as pareto optimal with respect to time and size.

Towards computing combinatorial objects, we described a parallel algorithm to measure the combinatorial coverage of any given covering array in the first part of this thesis. Based on our empirical results, parallel computing techniques carries big importance in counting missing t-tuples especially in higher strength values.

One of the main problem in metaheuristic search algorithms is need of efficient calculation techniques for the inner search. In the second part of this study, we tried to overcome this problem by developing novel parallel algorithms for the simulated annealing functions: neighbour state generation and fitness function. We presented an efficient technique to calculate the gain of neighbour state without measuring combinatorial coverage and also a parallel algorithm to generate valid neighbour states in the presence of constraints. These functions decreased the search time of simulated annealing, notably.

Besides parallelizing simulated annealing, we also described a way to run multiple simulated annealing concurrently in a way that, in each step of the inner search, several neighbour states are generated and the best of them which has the highest gain is chosen to move on with the next iteration. With this strategy, we decreased the negative effect of pure random search and increased the quality of accepted neighbour states in each iteration. This technique especially helped us to construct smaller size covering arrays and to cover the corner case t-tuples which are not easy to cover in the the presence of dense constraint structure.

Our novel hybrid approach reduces the size of covering arrays and search time further by decreasing the number of inner searches. Even though, our algorithm was not the best one in the empirical studies for both time and size, we can conclude that hybrid approach is pareto optimal.

In future work, we intend to modify these algorithms for other types of covering arrays such as cost & test-case aware covering arrays whose each configuration is associated with a set of test cases and testing cost can be specified at the granularity of option settings and test cases. Also, we plan to use a cloud of GPUs for the computations rather than a single one.

BIBLIOGRAPHY

[1] H. Avila George. *Constructing Covering Arrays using Parallel Computing and Grid Computing*. PhD thesis, 2012.

[2] H. Avila-George, J. Torres-Jimenez, V. Hernández, and N. Rangel-Valdez. A parallel algorithm for the verification of covering arrays. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications-PDPTA 2011*, pages 879–885, 2011.

[3] M. Banbara, H. Matsunaka, N. Tamura, and K. Inoue. Generating combinatorial test cases by efficient sat encodings suitable for cdcl sat solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 112–126. Springer, 2010.

[4] T. Berling and P. Runeson. Efficient evaluation of multifactor dependent system performance using fractional factorial design. *Software Engineering, IEEE Transactions on*, 29(9):769–781, 2003.

[5] R. C. Bryce and C. J. Colbourn. Constructing interaction test suites with greedy algorithms. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 440–443. ACM, 2005.

[6] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[7] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, 17(3):159–182, 2007.

[8] R. C. Bryce and C. J. Colbourn. One-test-at-a-time heuristic search for interaction test suites. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1082–1089. ACM, 2007.

[9] V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[10] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, 1997.

[11] M. B. Cohen, C. J. Colbourn, and A. C. Ling. Augmenting simulated annealing to build interaction test suites. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 394–405. IEEE, 2003.

[12] M. B. Cohen, M. B. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 121–132. IEEE, 2007.

[13] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139. ACM, 2007.

[14] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on*, 34(5):633–650, 2008.

[15] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 38–48. IEEE, 2003.

[16] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 58(121-167):0–10, 2004.

[17] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[18] J. Czerwonka. Pairwise testing: Available tools. `http://www.pairwise.org/tools.asp`.

[19] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. In *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, pages 419–430, 2006.

[20] S. R. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, pages 285–294. ACM, 1999.

[21] N. Francetić and B. Stevens. Asymptotic size of covering arrays: an application of entropy compression. *arXiv preprint arXiv:1503.08876*, 2015.

[22] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *Search Based Software Engineering, 2009 1st International Symposium on*, pages 13–22. IEEE, 2009.

[23] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering*, 16(1):61–102, 2011.

[24] S. Ghazi, M. Ahmed, et al. Pair-wise test coverage using genetic algorithms. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 2, pages 1420–1424. IEEE, 2003.

[25] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez. Construction of mixed covering arrays of variable strength using a tabu search approach. In *Combinatorial Optimization and Applications*, pages 51–64. Springer, 2010.

[26] M. Grindal, J. Offutt, and J. Mellin. Handling constraints in the input space when using combination strategies for software testing. 2006.

[27] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[28] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[29] A. Hartman. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms*, pages 237–266. Springer, 2005.

[30] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2-3):199–219, 2006.

[31] N. Ido and T. Kikuno. Lower bounds estimation of factor-covering design sizes. *Journal of Combinatorial Designs*, 11(2):89–99, 2003.

[32] R. Inam. An introduction to gpgpu programming-cuda architecture. *Mälardalen University, Mälardalen Real-Time Research Centre*, 2011.

[33] B. Jenkins. jenny: A pairwise testing tool. `http://www.burtleburtle.net/bob/index.html`, 2005.

[34] M. F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 46–55. ACM, 2012.

[35] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, et al. Optimization by simmulated annealing. *science*, 220(4598):671–680, 1983.

[36] N. Kobayashi. Design and evaluation of automatic test generation strategies for functional testing of software. *Osaka, Japan, Osaka Univ*, 2002.

[37] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 91–95. IEEE, 2002.

[38] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[39] R. E. Lopez-Herrejon, J. Javier Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, and E. Alba. A parallel evolutionary algorithm for prioritized pairwise testing of software product lines. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 1255–1262. ACM, 2014.

[40] T. Nanba, T. Tsuchiya, and T. Kikuno. Using satisfiability solving for pairwise testing in the presence of constraints. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 95(9):1501–1505, 2012.

[41] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[42] K. J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete applied mathematics*, 138(1):143–152, 2004.

[43] C. R. Rao. Factorial experiments derivable from combinatorial arrangements of arrays. *Supplement to the Journal of the Royal Statistical Society*, pages 128–139, 1947.

[44] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(3):277–331, 2004.

[45] P. J. Schroeder, P. Bolaki, and V. Gopu. Comparing the fault detection effectiveness of n-way and random test suites. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 49–59. IEEE, 2004.

[46] T. Shiba, T. Tsuchiya, and T. Kikuno. Using artificial life techniques to generate test cases for combinatorial testing. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 72–77. IEEE, 2004.

[47] N. J. Sloane. Covering arrays and intersecting codes. *Journal of combinatorial designs*, 1(1):51–63, 1993.

[48] B. Stevens, L. Moura, and E. Mendelsohn. Lower bounds for transversal covers. *Designs, codes and cryptography*, 15(3):279–299, 1998.

[49] K.-C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, (1):109–111, 2002.

[50] J. Torres-Jimenez and E. Rodriguez-Tello. Simulated annealing for constructing binary covering arrays of variable strength. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[51] J. Torres-Jimenez and E. Rodriguez-Tello. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1):137–152, 2012.

[52] Y.-W. Tung and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431–437. IEEE, 2000.

[53] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. In *Testing of Communicating Systems*, pages 59–74. Springer, 2000.

[54] A. W. Williams and R. L. Probert. Formulation of the interaction test coverage problem as an integer program. In *Testing of Communicating Systems XIV*, pages 283–298. Springer, 2002.

[55] H. Wu, C. Nie, F.-C. Kuo, H. Leung, and C. Colbourn. A discrete particle swarm optimization for covering array generation. 2012.

[56] J. Yan and J. Zhang. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, volume 1, pages 385–394. IEEE.

[57] C. Yilmaz. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on*, 39(5):684–706, 2013.

[58] C. Yilmaz, M. B. Cohen, A. Porter, et al. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[59] C. Yilmaz, S. Fouche, M. B. Cohen, A. Porter, G. Demiroz, and U. Koc. Moving forward with combinatorial interaction testing. *Computer*, (2):37–45, 2014.

[60] M. I. Younis and K. Z. Zamli. Mc-mipog: a parallel t-way test generation strategy for multicore systems. *ETRI journal*, 32(1):73–83, 2010.

[61] R. Yuan, Z. Koch, and A. Godbole. Covering array bounds using analytical techniques. *arXiv preprint arXiv:1405.2844*, 2014.