Developing multithreaded techniques and improved constraint handling for the tool CAgen

Michael Wagner, Manuel Leithner and Dimitris E. Simos
SBA Research
A-1040 Vienna, Austria
{mwagner,mleithner,dsimos}@sba-research.org

Rick Kuhn and Raghu Kacker
NIST, Information Technology Laboratory,
Gaithersburg, MD, USA
Email: {d.kuhn,raghu.kacker}@nist.gov

***Abstract*—CAgen is a state-of-the-art combinatorial test generation tool that is known for its execution speed. In addition, it supports an extensive list of features such as constraint handling, higher-index arrays, and import and export of models/test sets in various different formats. It is based on the FIPO algorithm, which can be considered an improved version of the widely used In-Parameter-Order strategy. In order to further speed up CAgen, this work first discusses how multithreading can be effectively used to optimally utilize available resources, particularly for large instances. We evaluate three different multithreaded variations of the horizontal extension and use the obtained insights to design the mFIPOG algorithm. In addition, we adopt methods that have previously been utilized to speed up constraint handling of CSP solvers in IPO algorithms into a forbidden tuple approach. In order to evaluate the performance of the improved tool, we provide results of benchmarks on the instances offered by the CT competition of IWCT 2022.**

I. Introduction

Combinatorial testing is a methodology that makes it possible to test large systems efficiently. It utilizes so-called t-way test sets, which, when executed against a system under test, can ensure that there is no fault triggered by a combination of up to t parameters of the input model [1]. This is ensured by the properties of *covering arrays* (CAs), the class of combinatorial designs underlying combinatorial test sets. A CA has the property that in every t-selection of columns, each possible t-tuple over the alphabets of the corresponding parameters has to appear in at least one row. In terms of testing, this means that each possible t-set of input combinations occurs in at least one test.

While combinatorial testing can significantly reduce the number of tests necessary to effectively test a system, generating such test sets with the smallest number of rows is not trivial in most cases. As the number of parameters of a system increases or if a higher strength t is deemed necessary for testing, the run time and memory demand during generation of such t-way test sets can rise exponentially. In order to keep up with the growing demand for such test sets, efficient test generation algorithms and tools are essential.

The research presented in this paper has been funded in part by the Austrian COMET K1 program. Moreover, this work was performed partly under the financial assistance award 70NANB21H124 from U.S. Department of Commerce, National Institute of Standards and Technology.

One such tool is the combinatorial test generation tool CAgen [2]. It is written in Rust[1], available online as a web GUI [2] and as a command line tool and has a rich set of features, such as multiple generation algorithms, efficient constraint handling and the possibility to generate test sets of higher index. While the web GUI enables users to easily create and edit models, generate and output test sets and help researchers and developers to become familiar with the concepts of combinatorial testing, the more powerful CLI version is ideal for complex problem instances. In this work, we focus on the CLI version of the tool for our discussions and benchmarks.

The algorithms implemented in this tool are based on the widely used In-Parameter-Order strategy for combinatorial test generation [3], which builds a CA incrementally by adding one column at a time by means of *horizontal extension*, while a *vertical extension* adds rows to the array in order to add any missing tuples if necessary, see Algorithm 1. The fastest algorithm implemented in CAgen is the FIPOG algorithm [4], owing its performance to various improvements on the algorithmic and implementation level. During the horizontal extension, FIPO iterates the rows of the CA in order and selects suitable values for the corresponding entries in the newly added column. This is done in a greedy manner by calculating the number of newly covered tuples, herein referred to as *coverage gain*, for each candidate value and selecting the value that maximizes this metric. In order to keep track of the coverage status of every possible t-tuple in each selection of t columns, a data structure called *coverage map* is used, which is explained in depth in [4].

This work is structured as follows. In Section II, we discuss multithreaded approaches that distribute the workload between multiple threads. Afterwards, Section III introduces methods and data structures that help speed up constraint handling in IPO algorithms using a minimal forbidden tuple approach. Section IV provides results based on the example benchmarks of the CT tool competition of IWCT 2022. Finally, we discuss potential future work in Section V.

[1]https://www.rust-lang.org/
[2]https://matris.sba-research.org/tools/cagen

Algorithm 1 IPOG Algorithm

procedure IPOG
  $Array \leftarrow$ cross-product of first t columns
  for $i \leftarrow t, \ldots, k$ **do**
    HorizontalExtension(i)
    if there are uncovered tuples **then**
      VerticalExtension(i)
    end if
  end for
end procedure

II. A MULTITHREADED FIPOG ALGORITHM

One way to speed up the run time of CA generation algorithms is to distribute the work between multiple threads. Finding an efficient way to parallelize the horizontal extension step, which usually takes up the majority of the execution time, is of particular interest. First, all selections of t columns need to be iterated in order to calculate the coverage gain for each candidate value. Second, after the candidate that maximizes the coverage gain for a row is selected, the coverage map needs to be updated accordingly.

Distributing these tasks efficiently between multiple threads without sacrificing the quality of the obtained solution is no trivial task. Two different approaches seem most suitable: We can either divide the workload by assigning a thread for each candidate value, or we can divide the coverage map into partitions and make different threads responsible for different sets of t-way selections of columns.

The former approach was explored by Younis and Zamli in [5], where they distributed the work of the horizontal and vertical extension between v_i threads, where v_i is the number of values the parameter at index i can assume. While the work proved that this strategy can provide a speedup for IPO algorithms, we decided against using it in this work. This is because one of the most impactful optimizations introduced in [4], particularly for higher strengths, is the precomputation of prefixes. In this step, the existing $(t-1)$ values are first used to calculate a base integer that can be added to any of the v_i different values, allowing for a fast lookup. This optimization would not be possible if different threads compute the scores for different values. Therefore, in this work, we focus exclusively on techniques that partition the coverage map. Three such approaches are described herein.

a) Variation 1 - Simplex setup: One approach where the coverage map is partitioned into several parts was recently explored in-depth in Antoine Veenstra's master thesis [6]. Veenstra introduces a parallel version of the horizontal extension step of an IPO algorithm, splitting the coverage map into partitions that can be read by different worker threads. During the horizontal extension, the worker threads compute the tuples that would be newly covered in their partition of the coverage map and send pointers to those tuples to the main thread, which then aggregates the results to select the best suitable candidate value and update the coverage map. Because

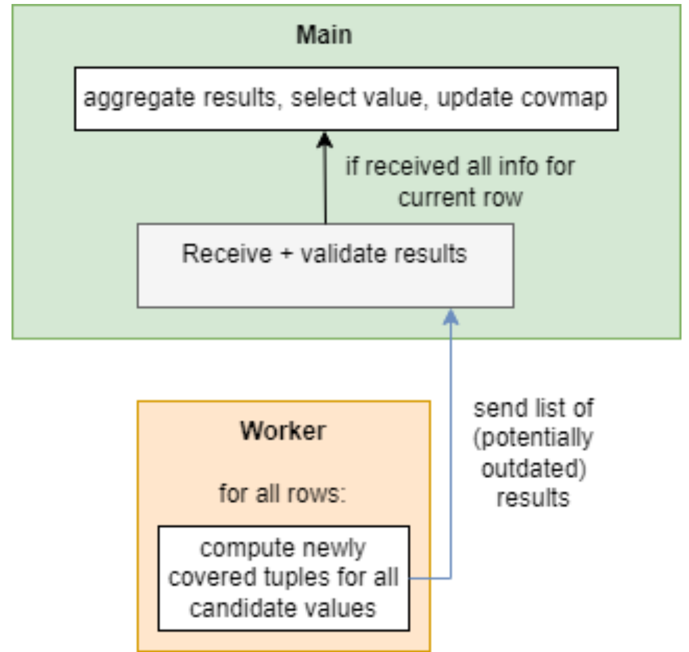

Fig. 1. Schematic of the multithreaded approach based on a simplex setup, introduced by Veenstra.

the coverage map might not be up to date while the worker threads calculate their scores, the main thread needs to validate whether the tuples represented by the received pointers are still uncovered. A schematic of the task distribution and the communication between the threads is shown in Figure 1. Since the task of each worker thread is identical (they just operate on different partitions of the coverage map), we only depict one worker thread in the graphic. Note that the worker threads do not wait for any feedback from the main thread, but simply continue computing the coverage gains for subsequent rows in respect to their partition of the coverage map.

A clear advantage of this approach is the minimal communication between the worker threads and the main thread. The approach is also quite sophisticated and considers many small details, such as rotating the partitions of the coverage map that a thread is responsible for with every new row to compensate for differences in run time due to *don't care* values. However, this technique has certain drawbacks. First, sending and collecting pointers to uncovered tuples can introduce significant overhead, especially in terms of memory consumption. This can become an issue when large instances are concerned. Further, since the main thread is tasked with aggregating and validating results as well as updating the coverage map, it can become a major bottleneck and lead to uneven work distribution when a larger number of worker threads is utilized.

To mitigate these flaws, we introduce two additional designs of a multithreaded horizontal extension based on partitioning of the coverage map. We compare the results with Veenstra's approach as well as the single threaded implementation of FIPOG.

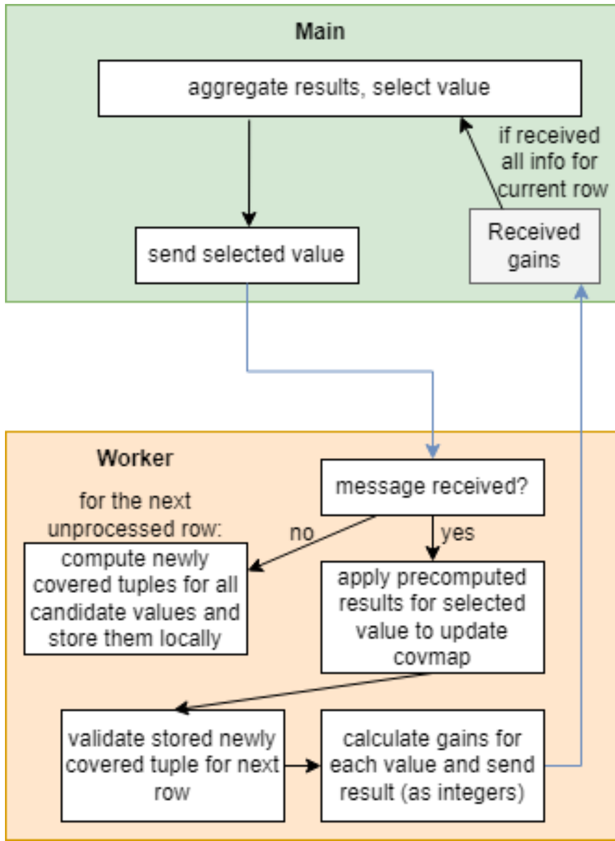

Fig. 2. Schematic of the multithreaded approach based on precomputed worker updates.

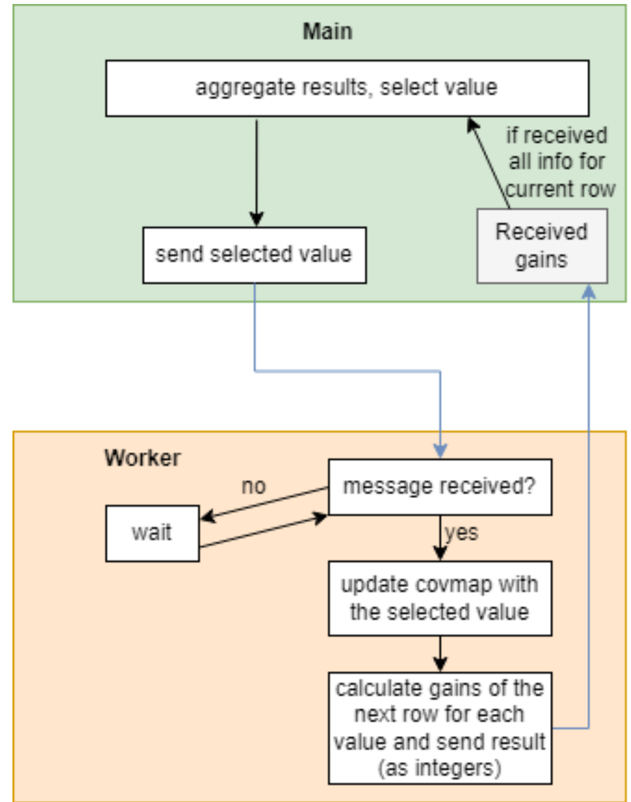

Fig. 3. Schematic of the multithreaded approach based on JIT worker updates.

b) Variation 2 - Precomputed worker updates: The first variation of the algorithm delegates the task of updating the coverage map to the worker threads, thus preventing the main/aggregation thread from becoming a bottleneck. Figure 2 visualizes the general workflow. Just like in the original approach, the worker threads precompute the newly covered tuples for all values despite a potentially outdated coverage map. However, instead of sending the results to the main thread for validation, they are stored locally by each thread. Whenever the main thread selects a value and notifies the workers about the decision, each worker thread first updates the coverage map based on this decision, utilizing the previously computed list of newly covered tuples. Then, the worker thread updates its results for the next row (the uncovered tuples that could be covered by each candidate value) based on the updated coverage map and sends the coverage gain for each candidate value to the main thread. While this approach distributes the work more evenly between worker threads, it is also more dependent on the communication delay between threads. Further, since each worker thread updates its own part of the coverage map, it is not safe to cycle to different partitions of the coverage map for each new row such as discussed in the first variation.

c) Variation 3 - Just-in-time (JIT) worker updates: Since the second variation maintains the same method of precomputing and storing newly covered tuples, the potential drawback of memory (and potentially run time) overhead still persists. Therefore, we introduce a third, seemingly naive, variation that entirely forgoes any precomputation. Figure 3 provides an overview of this process flow. The worker threads simply compute the coverage gain for their partition of the coverage map and send the results to the main thread (as integers). The main thread aggregates the results, selects a candidate value and finally notifies the worker threads about the decision. Instead of continuing to work on subsequent rows (as is the case for the previous variations), the worker threads wait for the response of the main thread. While this sounds highly inefficient in theory, large problem instances tend to make the computation of coverage gains more expensive, thus decreasing the impact of delays in communication.

d) Experimental evaluation: The compare the strengths and weaknesses of the different approaches presented in this work, we perform an extensive set of experiments, generating uniform CAs with 2, 5 and 10 values and 20, 30, 50 and 100 columns respectively. When a multithreaded algorithm is applied, the same variation of horizontal extension is used throughout all extension steps. All experiments were conducted on an Intel Xeon E3 processor with 8 logical processors and 64GB of RAM and the algorithm was allowed to spawn either 3 or 6 worker threads. The results of the benchmarks are displayed in Table I, where the time in milliseconds and the speedup of the different multithreaded variations compared

to the singlethreaded implementation is presented. A speedup greater than one refers to a faster execution time than the single threaded implementation.

While some of the results are quite expected, others are definitely a bit surprising. First, for small instances the single threaded implementation performs significantly better than all multithreaded variations. Since the run time for those instances is so small, the overhead to start new threads as well as communicate between them dwarfs any potential run time gains. One interesting thing to note is that each of the tested configurations performed the best in at least one of the tested instances. The Simplex variation seems to perform particularly well for medium-sized binary instances, but does not appear to scale well with an increased number of threads. The Simplex configuration with 6 threads was the only multithreaded variation that was slower on average than the single threaded implementation with an average speedup of 0.8, while also only performing better than the configuration with 3 threads in a small set of large instances.

The variation using precomputed worker updates performs a bit better on average, in particular for larger instances/alphabets. Furthermore, the configuration using 6 threads achieves a higher average speedup than the configuration with 3 threads while consistently terminating faster for larger instances. The biggest surprise was definitely the performance of the JIT worker update variation. While the approach appears incredibly wasteful in theory, the experiments proves that it can be quite potent in practice. Not only does it appear to scale well with an increased number of threads, it performed by far the best out of the tested configuration for the more difficult instances.

We believe that this first experimental evaluation nicely showcases that not only a lot more research is required to figure out the optimal multithreading techniques to speed up IPO algorithms for different instances, but also shows that sometimes approaches that appear to be too naive are worth taking in consideration.

III. Faster constraint handling

When modeling real-world systems, one must often consider special relationships and dependencies between components of the system. The ability to manually specify such relationships between parameters via constraints is therefore a crucial feature for any combinatorial test generation tool. For a comprehensive survey discussing the state of the art in this area of research, we refer the interested reader to [7].

Aside from the use of SAT or CSP solvers, the *minimum forbidden tuple* (MFT) approach has proven to be very effective for CA generation with IPO algorithms. CAgen implements the forbidden tuple approach as follows: First, a set of initial forbidden tuples is derived from a set of logical formulas. Then, a set of MFTs is computed, where each MFT can be represented as a list of parameter/value assignments. They are minimal in the sense that if any parameter/value assignment were removed from the tuple, the remaining tuple would be valid. Thanks to this property, one just has to ensure that there exists no row where all parameters that occur in some MFT take the values described by the MFT. For a more in-depth explanation of the MFT approach as well as the implementation details on how they are derived in CAgen, see [8] and [2].

Example 1: Assume we are given constraints in the form of the logical formula

$$(Par1 = true \ \& \ Par3 = false) => Par8 \neq 5 \qquad (1)$$

, which specifies that if parameter 1 is true and parameter 3 is false, then parameter 8 must not be equal to 5. In the form of forbidden tuples, this can be written as the set of parameter/value assignments [(1,true), (3,false), (8,5)].

In order for constraint handling techniques to be as efficient as possible, it is of upmost importance to avoid unnecessary validity checks during generation. In [9], the authors observed a significant redundancy when checking the validity of candidate values in both IPO extension steps. By using constraint groups to identify constraints that are irrelevant to the current parameter, they are able to significantly reduce the number of validity checks and thus the run time of the algorithm. In this work, we show that these observations and ideas can be easily integrated into a MFT approach.

The naive approach to confirm the validity of a candidate value during horizontal extension would iterate all minimal forbidden tuples and check if any of them occur in the current row. Of course, this leads to a large number of unnecessary validity checks, since only MFTs that contain the current parameter and the current candidate value are relevant. Additionally, out of this subset of MFTs, only those where all other parameters were already processed during earlier extension steps are of interest. Due to the minimality of MFTs, a MFT that contains parameters for which no value has been selected yet can not be violated. In order to perform validity checks in the most efficient way, we introduce a new data structure, called *one-directional MFT map* (o-map), which is generated in a preprocessing step. It is a ragged array that groups MFTs by the last parameter in the MFT and its value. For instance, assuming the parameters in Example 1 are processed in order, the prefix of the considered MFT, [(1,true), (3,false)], would be accessible at o-map[8][5], where o-map refers to the one-directional MFT map, 8 is the index of the last processed parameter and 5 the index of the corresponding value. Figure 4 visualizes this example. The one-directional MFT map thus contains every MFT exactly once. To determine the validity of a candidate value v_i during the horizontal extension of column i, it is sufficient to check if any of the parameter/value assignments specified in o-map[i][v_i] occur in the current row.

A similar reduction in the number of validity checks can be achieved during the vertical extension. Whenever a tuple is merged into an existing row by means of replacing existing *don't care* values, the algorithm needs to confirm that the replaced values do not violate any constraints. This can be checked efficiently by using a *bi-directional MFT map* data structure, which is similar to one introduced previously. The only difference is that in the bi-directional map, we not only

instance	t	rows	ST	Simplex - 3		Simplex - 6		precomputed - 3		precomputed - 6		JIT - 3		JIT - 6	
			time	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
2^{20}	2	12	**0**	7	0	8	0	23	0	59	0	21	0	78	0
	3	28	**0**	5	0	4	0	7	0	52	0	23	0	50	0
	4	66	**3**	14	0.21	30	0.1	9	0.33	55	0.05	19	0.15	50	0.06
	5	164	26	24	1.08	57	0.45	**22**	**1.18**	49	0.53	25	1.04	44	0.59
2^{30}	2	12	**0**	18	0	6	0	35	0	99	0	31	0	152	0
	3	36	**1**	6	0.16	14	0.07	42	0.02	78	0.01	35	0.02	93	0.01
	4	81	**21**	26	0.80	77	0.27	34	0.61	92	0.22	34	0.61	96	0.21
	5	206	259	**105**	**2.46**	230	1.12	119	2.17	151	1.71	144	1.79	154	1.68
2^{50}	2	14	**1**	13	0.07	31	0.03	56	0.01	152	0.01	59	0.01	150	0.01
	3	44	**6**	30	0.2	29	0.20	57	0.10	176	0.03	62	0.09	141	0.04
	4	101	201	**109**	**1.84**	237	0.84	131	1.53	217	0.92	130	1.54	212	0.94
	5	262	4195	1414	2.96	2382	1.76	1507	2.78	**1315**	**3.19**	1829	2.29	1580	2.65
2^{100}	2	16	**2**	31	0.06	54	0.03	105	0.02	298	0.01	120	0.01	270	0.01
	3	52	**53**	90	0.58	122	0.43	140	0.37	315	0.16	146	0.36	320	0.16
	4	132	3695	**1343**	**2.75**	2511	1.47	1486	2.48	1344	2.74	1645	2.24	1515	2.43
	5	337	183620	63275	2.90	100048	1.83	66134	2.77	**55023**	**3.33**	76968	2.38	63234	2.90
5^{20}	2	51	**1**	4	0.25	8	0.125	20	0.05	46	0.02	19	0.05	46	0.02
	3	393	**5**	34	0.14	85	0.05	30	0.16	44	0.11	29	0.172413793	51	0.09
	4	2698	138	174	0.79	272	0.50	131	1.05	146	0.94	**91**	**1.51**	123	1.12
	5	17245	3099	2093	1.48	2423	1.27	1414	2.19	1383	2.24	1245	2.48	**1164**	**2.66**
5^{30}	2	59	**2**	17	0.11	37	0.05	26	0.07	93	0.02	36	0.05	60	0.03
	3	466	**20**	62	0.32	140	0.14	43	0.46	108	0.18	40	0.5	117	0.17
	4	3390	935	619	1.51	879	1.06	489	1.91	453	2.06	417	2.24	**414**	**2.25**
	5	22469	36765	19352	1.89	20026	1.83	15035	2.44	12354	2.97	14009	2.62	**11278**	**3.25**
5^{50}	2	68	**3**	38	0.07	112	0.02	48	0.06	141	0.02	64	0.04	165	0.01
	3	590	147	172	0.85	346	0.42	127	1.15	235	0.62	**122**	**1.20**	211	0.69
	4	4486	13673	5334	2.56	6024	2.26	4898	2.79	**4031**	**3.39**	5203	2.62	4305	3.17
	5	30064	2054833	727760	2.82	**617291**	**3.32**	790018	2.60	726232	2.82	914433	2.24	839968	2.44
5^{100}	2	81	**6**	95	0.06	271	0.02	115	0.05	313	0.02	110	0.05	311	0.02
	3	762	1676	1005	1.66	1615	1.03	927	1.80	870	1.92	860	1.94	**804**	**2.08**
	4	5963	349688	135021	2.58	167782	2.08	151505	2.30	121846	2.86	132645	2.63	**105694**	**3.30**
10^{20}	2	229	**7**	25	0.28	47	0.14	29	0.24	76	0.09	15	0.46	48	0.14
	3	3475	156	274	0.56	415	0.37	162	0.96	165	0.94	**96**	**1.62**	129	1.20
	4	48712	7674	5491	1.39	6951	1.10	4222	1.81	3777	2.03	3190	2.40	**2870**	**2.67**
	5	614570	494720	270371	1.82	256519	1.92	245641	2.01	201908	2.45	227525	2.17	**187900**	**2.63**
10^{30}	2	247	**4**	65	0.06	114	0.03	42	0.09	107	0.03	48	0.08	97	0.04
	3	4146	496	687	0.72	868	0.57	462	1.07	409	1.21	**302**	**1.64**	322	1.54
	4	60445	52295	24722	2.11	29174	1.79	23034	2.27	19052	2.74	19925	2.62	**15499**	**3.37**
10^{50}	2	274	**11**	102	0.10	173	0.06	81	0.13	144	0.07	64	0.17	152	0.07
	3	5023	2772	2104	1.31	2834	0.97	1803	1.53	1475	1.87	1258	2.20	**1101**	**2.51**
	4	75692	778016	331523	2.34	313638	2.48	344416	2.25	265265	2.93	304549	2.55	**250180**	**3.10**
10^{100}	2	308	**77**	314	0.24	577	0.13	164	0.46	383	0.20	127	0.60	364	0.21
	3	6259	26728	12655	2.11	15113	1.76	11900	2.24	9889	2.70	10031	2.66	**8329**	**3.20**

TABLE I

BENCHMARKS FOR THE THREE MULTITHREADED VARIATIONS: SIMPLEX SETUP (SIMPLEX), PRECOMPUTED WORKER UPDATES (PRECOMPUTED) AND JUST-IN-TIME WORKER UPDATES (JIT). A SINGLETHREADED (ST) IMPLEMENTATION IS USED AS A REFERENCE. THE NUMBER NEXT TO THE ALGORITHM SPECIFIES THE NUMBER OF WORKER THREADS SPAWNED IN EVERY HORIZONTAL EXTENSION STEP, RUN TIME IS REPORTED IN MILLISECONDS.

variation	workers	avg. speedup	#best
ST	1	0	20
Simplex	3	1.08	3
	6	0.80	1
Precomp	3	1.13	1
	6	1.17	3
JIT	3	1.21	4
	6	1.25	11

TABLE II
SUMMARY OF MULTITHREADED EXPERIMENTS

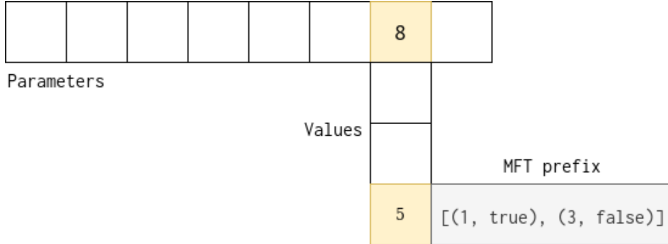

Fig. 4. The one-directional MFT map helps reduce the number of redundant validity checks.

add a MFT at the index defined by the last parameter/value in the MFT, but into all indices defined by any parameter/value in the MFT. This is necessary for the vertical extension, as values can be assigned to parameters that have already been processed in earlier extension steps, rendering the one-directional MFT map insufficient. While MFTs generally occur more than once in the bi-directional MFT map, it still helps to reduce the number of redundant validity checks significantly. A set of experiments using all implemented optimizations can be found in Section IV.

IV. Benchmark for the CT tool competition of IWCT 2022

For the CT tool competition of IWCT 2022 we submit the multithreaded mFIPOG algorithm. In order to perform decently for most instances the algorithm changes between the singlethreaded FIPOG implementation of the horizontal extension and the multithreaded JIT worker update variation described in Section II, depending on the number of t-way selections of columns, the strength t of the test set and the number of candidate values. With this combination we hope to combine the unmatched performance of single threading for small instances with the multithreaded approach that seemed most promising in terms of scalability. We plan to integrate the mFIPOG algorithm officially into the tool CAgen after the competition.

Table III shows the results for all benchmark instances provided for the competition that do not contain any constraints. Due to the relatively small instances and in particular the small number of columns the algorithm mostly relies on the singlethreaded horizontal extension while the multithreaded variation helps speed up some of the strength 4 instances.

We also attempted to solve some of the benchmarks with constraints, but did not manage to complete most of them. The minimal forbidden tuple approach for constraint solving works very well for real-world systems with a moderate

instance	t	rows	time(ms)
UNIFORM_BOOLEAN_0	2	10	0
	3	24	0
	4	56	0
UNIFORM_BOOLEAN_1	2	10	0
	3	22	0
	4	54	0
UNIFORM_BOOLEAN_2	2	10	0
	3	22	0
	4	54	0
UNIFORM_BOOLEAN_3	2	12	0
	3	26	0
	4	66	2
UNIFORM_BOOLEAN_4	2	8	0
	3	18	0
	4	38	0
UNIFORM_ALL_0	2	146	0
	3	1744	45
	4	19405	866
UNIFORM_ALL_1	2	368	2
	3	8369	152
	4	161537	6591
UNIFORM_ALL_2	2	51	0
	3	369	3
	4	2415	58
UNIFORM_ALL_3	2	333	0
	3	7175	12
	4	128739	905
UNIFORM_ALL_4	2	459	2
	3	11554	165
	4	250032	8093
MCA_0	2	2296	8
	3	116855	1382
	4	5397481	350192
MCA_1	2	408	0
	3	816	0
	4	1632	0
MCA_2	2	518	0
	3	4144	0
	4	8290	3
MCA_3	2	1366	1
	3	27744	67
	4	519257	5458
MCA_4	2	1426	1
	3	38392	42
	4	691407	2400

TABLE III
EXPERIMENTAL RESULTS OF THE mFIPOG ALGORITHM FOR THE BENCHMARK INSTANCES OF THE CT-TOOL COMPETITION OF IWCT 2022.

number of constraints where a lot of the validation work is reduced by means of preprocessing in the form of forbidden tuple computation. At the same time, this preprocessing step is not feasible anymore when confronted with such a large number of constraints as provided in the benchmark instances. Even for the lightest instance where the largest constraint specification consists of ”only” 21000 characters both the initial forbidden tuple derivation as well as the minimal forbidden tuple computation take more than 20 minutes in our implementation, while the MFT map generation and the actual test generation finish within seconds. In the future we plan to examine additional constraint solving techniques to supplement our existing approach.

V. Conclusion and Future Work

In this work, we discussed different multithreaded variation of the horizontal extension of FIPOG algorithms and provided an extensive set of experimental results to evaluate them and demonstrate that they a significantly speedup of the test generation process can be achieved. We made use of the

gained insights to design mFIPOG, an efficient multithreaded algorithm, which we tested on the benchmark instances for the IWCT 2022 CT tool competition. Last, we introduced a more efficient way of applying the minimal forbidden tuple approach to constrained CA generation.

In future work, we plan to investigate in more detail how different multithreading techniques perform when a larger number of threads is available as well as explore how high performance computing can help accelerate the FIPOG algorithm even further. We also want to speed up different parts of the generation process with parallel methods. Last, we want to integrate our novel CPHFIPO [10] and SIPO [11] algorithms into CAgen, preferably in a parallel variation.

Disclaimer: *Products may be identified in this document, but identification does not imply recommendation or endorsement by NIST, nor that the products identified are necessarily the best available for the purpose.*

References

[1] D. R. Kuhn, R. N. Kacker, Y. Lei *et al.*, "Practical combinatorial testing," *NIST special Publication*, vol. 800, no. 142, p. 142, 2010.

[2] M. Wagner, K. Kleine, D. E. Simos, R. Kuhn, and R. Kacker, "Cagen: A fast combinatorial test generation tool with support for constraints and higher-index arrays," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 191–200.

[3] Y. Lei and K. C. Tai, "In-parameter-order: a test generation strategy for pairwise testing," in *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, Nov 1998, pp. 254–261.

[4] K. Kleine and D. E. Simos, "An efficient design and implementation of the in-parameter-order algorithm," *Mathematics in Computer Science*, Dec 2017.

[5] M. I. Younis and K. Z. Zamli, "Mc-mipog: a parallel t-way test generation strategy for multicore systems," *ETRI journal*, vol. 32, no. 1, pp. 73–83, 2010.

[6] A. Veenstra, "Accelerating mixed-level coverage array generation," Master's thesis, University of Twente, 2021.

[7] H. Wu, C. Nie, J. Petke, Y. Jia, and M. Harman, "A survey of constrained combinatorial testing," *arXiv preprint arXiv:1908.02480*, 2019.

[8] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Constraint handling in combinatorial test generation using forbidden tuples," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–9.

[9] L. Yu, Y. Lei, M. Nourozborazjany, R. N. Kacker, and D. R. Kuhn, "An efficient algorithm for constraint handling in combinatorial test generation," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 242–251.

[10] M. Wagner, C. J. Colbourn, and D. E. Simos, "In-parameter-order strategies for covering perfect hash families," *Applied Mathematics and Computation*, vol. 421, p. 126952, 2022.

[11] M. Wagner, L. Kampel, and D. E. Simos, "Heuristically enhanced ipo algorithms for covering array generation," in *International Workshop on Combinatorial Algorithms*. Springer, 2021, pp. 571–586.