Automatic Scalable System for the Coverage-Directed Generation (CDG) Problem

Raviv Gal[1], Eldad Haber[2], Wesam Ibraheem[1], Brian Irwin[2], Ziv Nevo[1], Avi Ziv[1]
[1]Hybrid Cloud Quality Technologies Department, IBM Research - Haifa, Israel
{ravivg, wesam, nevo, aziv}@il.ibm.com
[2]Department of Earth and Ocean Science, The University of British Columbia, Vancouver, BC, Canada
{haber, birwin@eoas.ubc.ca}

***Abstract*—We present AS-CDG, a novel automatic scalable system for data-driven coverage-directed generation. The goal of AS-CDG is to find the test templates that maximize the probability of hitting uncovered events. The system contains two phases, one for a coarse-grained search that finds relevant parameters and the other for a fine-grained search for the settings of these parameters. To overcome the lack of evidence in the search, we replace the real target with an approximated target induced by neighboring events, for which we have evidence. Usage results on real-life units of high-end processors illustrate the ability of the proposed system to automatically find the desired test-templates and hit the previously uncovered target events.**

I. Introduction

Coverage closure is the process of advancing coverage goals in general, and coverage levels specifically [1]. Since coverage is one of the main quality indicators of the verification process and the *design-under-verification* (DUV), coverage status is an important criterion for many project milestones, such as tape-outs. As a result, the verification team can spend significant time and effort on coverage closure.

To achieve coverage closure, the verification team needs to analyze the uncovered events and understand what is needed to hit these events. Only then can they write or modify tests or test-templates (the input to the random generator) that hit the uncovered events or improve the probability of hitting them.

The verification environments of complex units can contain hundreds or thousands of parameters that affect the generated stimuli. When trying to hit a given coverage event, or a set of related events, the verification engineer first needs to identify the small set of parameters that most influence the environment's capacity to hit the desired events. Next, they need to find the optimal settings of these parameters that maximize the probability of hitting the events. These steps are time consuming and require an understanding of the target design and the verification environment. This makes coverage closure one of the bottlenecks of the entire verification process.

Coverage-Directed Generation (CDG) [2] is a generic name used for a multitude of techniques that create tests or test-templates for hitting uncovered events. This paper presents AS-CDG, a novel automatic CDG flow for large and complex DUVs. AS-CDG is based on exploring and exploiting the probabilistic relations between verification parameters and the target coverage events. This exploration is difficult because of the complete lack of positive evidence for any settings that hit the target events. To overcome this difficulty, we replace the real target with an approximated target induced by neighboring events (i.e., events that when hit increase the probability of the target event being hit as well), for which we have evidence.

AS-CDG contains two main phases that match the manual steps mentioned above. It starts with identifying those relevant parameters that most affect the coverage achieved. This is done using the *Template-aware coverage* (TAC) approach [3] to identify existing test-templates that best hit the approximated target. The second phase performs a fine-grained search in the space of the settings of the parameters identified in the first phase. The goal of this phase is to find the settings that maximize the probability of hitting the actual target events. This phase comprises two subphases: we begin with a random sampling of the space induced by the settings of the selected parameters. This is followed by a *derivative-free optimization* (DFO) technique [4] to find near-optimal settings.

The mapping from the settings of the parameters in a test-template to the coverage is unknown, making the use of analytical optimization techniques impossible. Hence, we rely on a DFO technique to find the optimal settings. Moreover, this mapping is probabilistic in nature because random stimuli generation can lead to different coverage results when simulating different test-instances generated from the same test-template. This probabilistic nature of the mapping can be viewed as dynamic noise, which the optimization technique must be able to handle. Our flow uses the optimization procedure described in [5]. Specifically, we use the implicit filtering [6] algorithm that is both simple and effective.

The implementation of the proposed flow utilizes a suite of tools. TAC [3] identifies existing test-templates with relevant parameters. These test-templates are fed to a *Skeletonizer* that identifies the settings for the fine-grained search and creates a skeleton of a test-template that the CDG-Runner application can use to create valid test-templates during the search. The CDG-Runner application is responsible for the second phase of the flow. It creates test-templates that fit the skeleton according to the specific task it executes (e.g., random sample, optimize), sends the templates to the batch environment for simulation, collects the coverage data, analyzes the results, and decides on the next step.

The entire flow operates outside the existing design and verification environment and is "black box" in nature. Consequently, applying the flow does not require any changes to the

978-3-9819263-5-4/DATE21/©2021 EDAA

verification environment. The flow can be used "as is" in any verification environment that uses parametrized test-templates.

The flow described in the paper has been used in the verification of many units inside two high-end processor systems. In almost all cases, each step in the CDG flow contributed to the coverage by improving the number of hits for lightly-hit events and by hitting uncovered events. The few cases where the flow failed to provide the desired results occurred because the events were unhittable or because the verification environment lacked the required capabilities.

The rest of the paper is organized as follows: Section II surveys the history of CDG. Section III provides a detailed description of the CDG problem. Section IV, the main section of the paper, describes the proposed flow in detail. Usage results are given in Section V; Section VI concludes the paper.

II. Related Works

Since the emergence of *Coverage-Driven Verification* (CDV) methodologies [1], the search for automatic techniques that assist in coverage closure has become one of the holy grails of hardware functional verification. This made Coverage-Directed Generation (CDG) a popular research topic that received significant attention both in academia and industry.

Early CDG papers showed encouraging results, however, none of them matured into an industrial solution. This is due to scalability issues, usage complexity and having DUV specific components that limit the generality of the solution. In general, approaches for CDG can be classified into two main categories, model-based CDG ([7], [2]) and data-driven CDG ([8], [9], [10], [11]). In model-based CDG, a model of the DUV is used to generate test-instances or test-templates. Mishra et al. [7] used an architecture model, while Ur et al. [2] used a micro-architecture model. Both techniques failed to scale due to limitations of formal methods. In addition, these techniques require the model definition to be relatively accurate in order to hit hard events, making its definition a complex task that requires high maintenance during project lifetime.

In data-driven CDG, the system discovers and learns the complex relations between the test-template parameters and the coverage events. Wagner et al. [8] used Markov chains to capture the relationship, where the chain is design and domain specific. Smith et al. [10] used a genetic algorithm approach to generate new test instances. A major problem in this method is handling the validity of the evolutionary tests. Fine et al. [11] used Bayesian Networks to guide the input generation. While the network weights are learned automatically, the network topology is DUV specific and requires domain knowledge. The proposed system, on the other hand, is DUV independent and fully automated, overcoming these obstacles.

Recent works on CDG have limited their goals. Instead of seeking to generate new test-templates, they have sought to select the best test-templates or test-instances to simulate, based on previous runs. Yang et al. [12] removed test-templates that do not contribute to coverage. Gal et al. [3] collected statistics over the coverage hit by each test-template, and used them to automatically suggest a regression policy that focuses on events hardly hit. Wang et al. [13] used machine learning to learn the relationship between transactions and coverage, and filter test instances accordingly. Laeufer et al. [14] used fuzzing techniques from the software testing realm to mutate existing test instances. Their technique requires changes in the DUV and has limitations regarding the validity of the tests created.

Recently, Gal et al. [5] presented an optimization-based approach to CDG. Their approach searches for the best test-template to hit uncovered events using a derivative-free optimization algorithm. This paper extends this work and shows how it can be used in real-life settings with large DUVs.

III. Problem Description

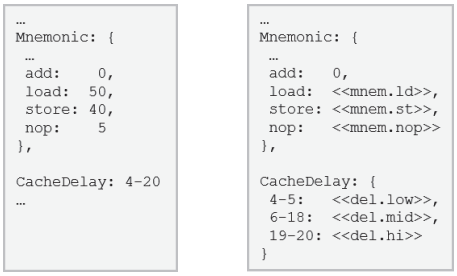
This work targets verification environments that use a biased random stimuli generator to generate stimuli. It is common for the verification environments of large and complex DUVs, such as units of high-end processors, to contain hundreds or even thousands of different parameters that control the stimuli generation. When the verification team needs to exercise certain areas and features in the DUV, they create *test-templates* that modify the default settings for a set of parameters, while leaving the rest of the parameters to their default behavior. These test-templates are used as the input for the stimuli generator.

Identifying the relevant parameters and finding the correct settings of these parameters can be a tedious manual process that requires deep understanding of the DUV and its verification environment. In many cases, the process is a trial-and-error process that requires several iterations before a good test-template is found. This work describes a system and a flow that can, in many cases, automate the process.

We use two types of parameters: range parameters and weight parameters. A range parameter is simply a range of values. When the stimuli generator needs to make a random decision related to a range parameter, it uniformly selects a value from the range. A weight parameter is a set of value-weight pairs. When the stimuli generator needs to make a random decision related to a weight parameter, it uses the weights as a distribution function for selecting a random value.

Figure 1(a) shows a snippet of a test-template for stressing the load store unit of a processor with a weight parameter for the instruction mnemonic and a range parameter for the cache delay. Note that parameters can be used many times during the generation process, and the number of times a parameter is used may differ from parameter to parameter and per test-instance. For example, the `mnemonic` parameter is used for every instruction generation, while `CacheDelay` is used only when the cache is accessed.

Simulating a test-instance on the design produces a coverage vector, indicating whether each coverage event was hit in this simulation. Due to the randomness of the stimuli generator, test instances originating from the same test-template may produce different coverage vectors. A summary of the coverage vectors created by all the test-instances is stored in a coverage repository. During coverage closure, the verification team queries the coverage repository to find important uncovered events. They can then create test-templates that have a high probability of hitting these events. This work presents an automatic flow that creates such templates automatically.



(a) A test-template snippet (b) The resulting skeleton

Fig. 1. Example of a test-template and the skeleton it induces

IV. AS-CDG

The goal of AS-CDG, our coverage-directed generation system, is to find a test-template that maximizes the probability of hitting uncovered events. The system is data-driven, meaning it is based on data collected during executions of the DUV. There are three main challenges that the system must address to achieve its goal: the lack of evidence for tests hitting the target event, the large number of parameters, and the probabilistic nature of the mapping from parameter space to coverage space.

Figure 2 depicts the main components of AS-CDG and the flow it implements. The rest of this section provides more details on each of the main components and steps in AS-CDG.

A. Approximated Target

One of the hurdles in applying data-driven techniques for CDG is the complete lack of positive evidence for any test-template that has a non-zero probability of hitting the target event. This means that any search or optimization technique needs to search in the dark to find a good starting point for its operation. To overcome this problem, we define an approximated target. This approximated target is based on the coverage of events that are near the target events. The idea, which mimics the work of verification experts, is that by improving the probability of hitting these neighbors, we exercise the relevant area in the DUV. This, in turn, increases the probability of hitting the target event itself. The validity of using approximated targets is backed-up by the usage results.

There are many possible ways to automatically find the neighbors of a coverage event. For example, Wagner et al. [8] used the natural order of buffer utilization to learn how to fill a buffer. Fine and Ziv [15] exploited the structure of a cross-product coverage model. Gal et al. [16] used formal methods to find a set of neighbor events with positive and negative information regarding the probability of a test hitting the target event. We use all these methods in AS-CDG and demonstrate their effectiveness in many cases.

The approximated target function can be the sum of the neighbor and target events, or a weighted sum of these events, giving more weight to events closer to our target.

B. Coarse-Grained Search

The large number of parameters used in a verification environment, and the high level of noise in the relationship between the parameters and coverage, mean that performing the search for the optimal parameter settings on the entire parameter set is either infeasible or would require a very large number of simulations. Therefore, we perform the search in two phases. We begin with a coarse-grained search that identifies the relevant parameters with the most influence on the probability of hitting the target. Only then do we perform a search for the optimal settings of these parameters.

The search for relevant parameters focuses on existing test-templates. The verification environment of a typical DUV contains many test-templates that were developed by the verification team to address all sorts of verification needs. Each of these test-templates is used to generate many test-instances that are simulated on the DUV. First order statistical analysis of the coverage achieved by test-instances generated from each of the templates can easily reveal the test-templates that best hit the (approximated) target. Because these test-templates probably contain the parameters that are most relevant for hitting the target, those parameters are the ones upon which the fine-grained search should focus.

Template-Aware Coverage [3] is a tool that maintains first-order statistics on the coverage of each event by each test-template. The statistics include the probability of hitting the event with a test instance generated from the test-template. The TAC tool was designed to answer the type of queries needed for the coarse-grained search. Namely, given a list of the neighbor events of the target, find the best n test-templates that hit these events. The parameters in these test-templates are selected to be the ones used in the fine-grained search.

C. Skeletonizer

The coarse-grained search results in test-templates that contain those parameters relevant for hitting the target events. The goal of the fine-grained search is to find the best settings of these parameters that maximize the coverage of the target events. To perform this task, we first need to identify the settings. We developed a *Skeletonizer* that receives a test-template as its input and produces a skeleton of the test-template with all the settings that can be set by the CDG-Runner marked.

The operation of the Skeletonizer is fairly simple. It parses the input test-template, identifies parameters in the file that can be handled by the CDG-Runner, and marks the settings that can be changed by it. The output of the tool is a skeleton file that looks like the original test-template, except for the places that can be modified by the CDG-Runner.

Specifically, the Skeletonizer handles the types of parameters described in Section III as follows. For the weight parameters, the tool simply identifies the weights in the parameter and replaces them with a mark. The top of Figure 1(b) shows the skeleton of the `Mnemonic` parameter. Note that the weight for `add` was not marked. This is done intentionally, because zero weights often indicate values that should not be used. The user has the option to include zero weights in the markings.

Range parameters, from which values are selected uniformly, are replaced with weight parameters, as shown in the bottom halves of Figure 1(a) and (b). The full original range is replaced with smaller subranges, each with its own weight. This allows the CDG-Runner to better control the distribution

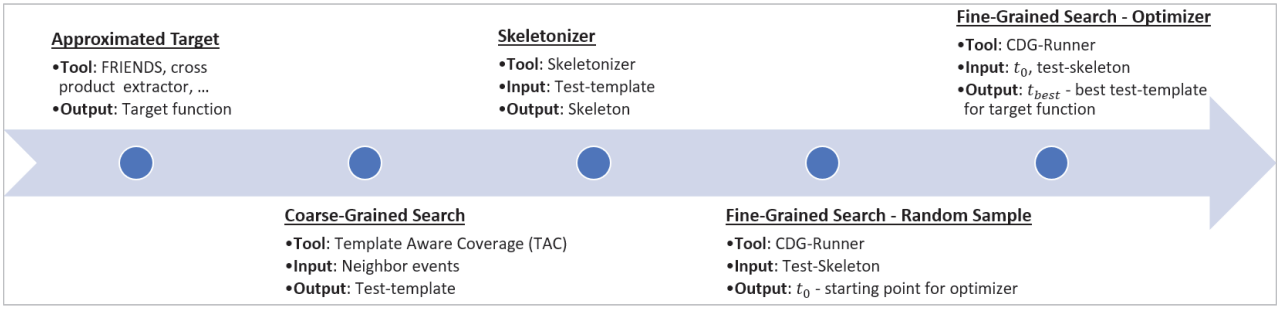


Fig. 2. AS-CDG flow

of the parameter. In the example presented in the figure, setting high weights for the low subrange and low weights for the other subranges would lead to shorter delays in cache response. The user can control the number of subranges used and how they span the entire range.

D. Random Sample

After the Skeletonizer marked the weights, the CDG-Runner can search for the optimal settings that maximize the probability of hitting the target. We start this search by performing a random sample of the parameter settings and measuring an estimate for the approximated target. In the random sampling process, the skeleton created by the Skeletonizer is used to create n random test-templates that uniformly span the weights in the skeleton. We then generate and simulate N test-instances from each template. The coverage obtained from the simulations is then used to estimate the probabilities of hitting each event e by each template t according to $e_N(t) = \frac{1}{N}\sum_i s_i^e(t)$, where $s_i^e(t)$ is an indicator of whether test-instance i generated from test-template t hits event e. With this, we can calculate the approximated target $T_N(t) = \sum_{e \in E} e_N(t)$, where E are the neighbors of the target events.

The random sample requires $n \times N$ simulations, and while its probability of hitting the actual target is low, it helps find a good starting point for the subsequent optimization step. Specifically, the optimization step can begin with the test-template that reaches the highest target value. This good starting point can save the optimization algorithm many iterations of wandering in an almost flat area reached by a random start. This makes the investment of $n \times N$ simulations worthy.

E. Optimizer

At the heart of AS-CDG lies an optimization algorithm that searches for test-templates that maximize the probability of hitting each of the target events. While casting CDG as an optimization problem is natural, there are a number of challenges that complicate the use of optimization techniques for CDG. The first challenge is the nature of the objective function. As explained earlier, we do not have access to the objective function directly. Instead, we must rely on an estimate of the function obtained from simulations. This leads to unknown dynamic noise in the observed objective value. Therefore, we cannot apply commonly used optimization methods that rely on first-order derivatives (gradient methods) and second-order derivatives (Hessian methods) of the objective function. To address this challenge, we rely on *derivative free optimization* (DFO) methods [4]. These methods require only samples of the objective function itself, without the need to calculate its derivatives. Specifically, we use the *implicit filtering* algorithm that was proven to be efficient in CDG settings [5].

Algorithm 1 Implicit filtering algorithm

```
procedure IF(n, N, h, t_0, stopping criteria)
    repeat
        best ← T_N(t_0)
        next_center ← t_0
        D ← set of n random directions
        for all directions d in D do
            t ← (t_0 + d × h)
            if T_N(t) > best then
                best ← T_N(t)
                next_center ← t
        if next_center == t_0 then
            h ← h/2
        t_0 ← next_center
    until stopping criteria is met
    return t_0
```

Algorithm 1 describes the implicit filtering algorithm. The algorithm starts with t_0, the best random sample from the random sampling step. At each iteration of the algorithm, it selects n random directions and samples the objective function at points with distance h (a.k.a. step size or stencil size) from the current center in each of the selected directions. If the best value of these samples is better than the value at the center, the center is moved to that point and the process repeats. Otherwise, when the best result is at the center, the distance h is halved and the process repeats. This is done to reduce the possibility of overshooting the maximum. The algorithm stops when a stopping criterion is met. The stopping criteria is usually a combination of the number of iterations, the current stencil size value, and the hit probability of the target event.

The implicit filtering algorithm has several hyperparameters: n, the number of directions used in each iteration; h, the initial stencil size; and the stopping criteria. Each of these hyperparameters can affect the convergence rate of the algorithm in terms of iterations and number of samples.

When dealing with dynamic noise, we make two small modifications to the base algorithm. First, we use another hyperparameter N, the number of samples per point. Increasing N reduces the effective noise and thus can lead to faster convergence. On the other hand, increasing N increases the number of samples needed per iteration. It's also a common practice to resample the center point in each iteration, even though it was sampled in the previous iteration. This resampling is used to reduce the effect of extremely high noise.

At each iteration of the implicit filtering algorithm, it creates $n+1$ test-templates, one for each of the n random directions and one for the center. Each of these templates is simulated N times and the empirical expectation $e_N(t)$ for each of the events is calculated. This result is used to calculate an estimation for the approximated target, which is in turn used to calculate the starting point for the next iteration. The output of the algorithm is the best template found in the last iteration. Once there is good evidence for the target event, we can repeat the process, this time with the real objective function.

F. Harvesting the Best Test-Template

Once the optimizer finishes its work, we harvest the test-template that has the highest probability of hitting the target events. This test-template is added to the regression suite of the DUV to ensure that the target events will continue to be hit often and on a regular basis.

V. Deployment Results

Our AS-CDG solution was deployed in many real-life hardware verification projects, significantly helping their coverage-closure effort. It enabled hitting hundreds of uncovered events, while improving hit rates of neighboring events. All these projects were verifying large units in our IBM's high-end processors. A unit in these processors is a rather large and complex piece of logic with typically tens of thousands of coverage events. This section presents the deployment results from three units: an I/O unit, an L3 cache, and an Instruction Fetch Unit (IFU). Because AS-CDG is not comparable to recent CDG papers and older work cannot scale to the size of these units and/or is not available, we compare the results of using AS-CDG with the results when a manual process is used.

For each processor unit, we identified hard-to-hit events, focusing on those belonging to a larger family of events, e.g., filling-a-buffer events or a cross-product.

We defined our approximated target function as the sum of the hit counts for all the events in the family. In this way, when the events in a family are truly related and when the family contains a descent gradient from easily hit events to hard-to-hit events, the optimization process is likely to converge quickly.

Finally, we applied the rest of the AS-CDG flow. We also repeatedly simulated the best test-template found to assess its quality. Our results for two units, an I/O unit and an L3 cache, are shown in Figures 3 and 4 respectively. The tables show, for each unit, hit counts and hit rates at the various phases for a given family of coverage events. The color coding follows IBM's convention where an event with a hit count smaller than 100 is considered lightly hit and is colored in orange. In addition, we also consider an event with a hit rate smaller than 1% to be lightly hit. Never-hit results are colored in red. All other results (well-hit events) are colored in green.

Event name	Before CDG (669000 sims)		Sampling phase (200 tests x 100 sims each)		Optimization phase (7 iterations x 20 tests x 200 sims)		Running best test (10000 sims)	
	#hits	hit rate	#hits	hit rate	#hits	hit rate	#hits	hit rate
crc_004	69048	10.321%	2280	11.400%	7963	28.439%	9182	91.820%
crc_008	14736	2.203%	469	2.345%	7418	26.493%	8865	88.650%
crc_016	1213	0.181%	92	0.460%	6169	22.032%	8234	82.340%
crc_032	12	0.002%	10	0.050%	3890	13.893%	6659	66.590%
crc_064	0	0.000%	0	0.000%	716	2.557%	2832	28.320%
crc_096	0	0.000%	0	0.000%	12	0.043%	646	6.460%

Fig. 3. Hit statistics for a family of events in one of the I/O units

Event name	Before CDG (1,000,000 sims)		Sampling phase (210 tests x 100 sims each)		Optimization phase (25 iteration x 12 tests x 100 sims)		Running best test (15000 sims)	
	# hits	hit rate	# hits	hit rate	# hits	hit rate	# hits	hit rate
byp_reqs01	865,000	86.500%	18,109	86.233%	24,957	83.190%	14,478	96.520%
byp_reqs02	284,800	28.480%	10,015	47.690%	19,569	65.230%	13,038	86.920%
byp_reqs03	25,600	2.560%	4,425	21.071%	13,251	44.170%	9,608	64.053%
byp_reqs04	2,000	0.200%	3,109	14.805%	11,232	37.440%	8,409	56.060%
byp_reqs05	300	0.030%	2,558	12.181%	10,339	34.463%	7,989	53.260%
byp_reqs06	0	0.000%	2,052	9.771%	9,492	31.640%	7,536	50.240%
byp_reqs07	0	0.000%	1,598	7.610%	8,485	28.283%	7,026	46.840%
byp_reqs08	0	0.000%	1,242	5.914%	7,424	24.747%	6,387	42.580%
byp_reqs09	0	0.000%	903	4.300%	6,268	20.893%	5,559	37.060%
byp_reqs10	0	0.000%	640	3.048%	4,957	16.523%	4,537	30.247%
byp_reqs11	0	0.000%	389	1.852%	3,494	11.647%	3,401	22.673%
byp_reqs12	0	0.000%	193	0.919%	2,148	7.160%	2,245	14.967%
byp_reqs13	0	0.000%	90	0.429%	1,096	3.653%	1,180	7.867%
byp_reqs14	0	0.000%	33	0.157%	450	1.500%	462	3.080%
byp_reqs15	0	0.000%	6	0.029%	125	0.417%	127	0.847%
byp_reqs16	0	0.000%	0	0.000%	24	0.080%	15	0.100%

Fig. 4. Hit statistics for a family of events in a processor's L3 unit

The first two columns after the event name column show hit counts and hit rates before AS-CDG was applied. These are the results of applying mainstream unit simulation for several weeks, utilizing multiple test-templates. Using simulation statistics from TAC and expert advice, we selected a test-template with parameters that should best hit the family of events at hand. We then "skeletonized" this test-template and generated several hundreds of new mutated test-templates. Simulating each new test-template multiple times (typically 100 times) gave us the numbers in the third and fourth columns. The next two columns show hit counts and hit rates after running the *optimization* phase. The starting point for this phase is the best test-template from the sampling phase (the one with the highest score for the target function). Finally, the last two columns show the results of running the best test-template from the optimization phase multiple times. This was usually the test-template which the verification team chose to add to their daily regression.

It is evident from the results that the suggested flow improves both hit counts and hit rates for both families of events. Moreover, each phase improves upon its predecessor. Take the L3 unit results for example (Figure 4). Starting with 5 well-hit events and 11 never-hit events (looking at hit counts), the sampling phase alone, using just 21,000 simulations, is able to turn 7 uncovered events into well-hit events and 3 uncovered events into lightly-hit events. The optimization phase is able to then turn the 3 lightly-hit events into well-hit events and the remaining uncovered event into a lightly-hit event, using just 30,000 simulations. Finally, the best test-template shows significantly better hit rates.

Results for a third unit, an IFU, are shown in Figure 5. The figure shows the number of events in each status at each phase. This time, the events belonged to a family of 256 events,

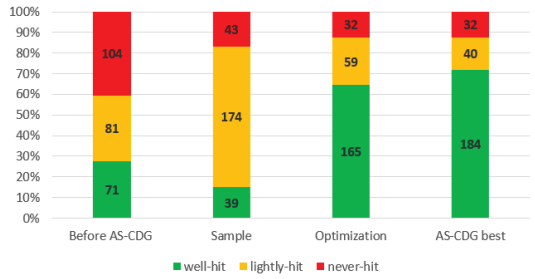


Fig. 5. Event status while running AS-CDG on a cross-product (IFU)

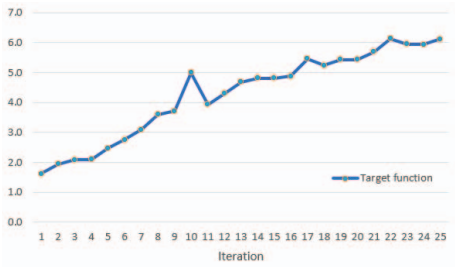


Fig. 6. Optimization progress on the L3 example

defined by a cross product of four features: entry (0–7), thread number (0–3), sector (0–3) and branch (0–1). Note how many uncovered events the sampling phase was able to hit, and how the optimization phase made most of the events well hit. Having said that, 32 events (all entry7 events) remained uncovered at the end of the flow, and are considered out of the unit capabilities to hit. This demonstrates that for cross products as well, AS-CDG is able to hit many previously-uncovered events, and improve hit counts and hit rates across the board.

Figure 6 shows the maximal value of the target function per optimization iteration. The data refers to the optimization phase on the L3 unit. It can be seen that the optimization process makes a gradual progress towards a (local) maximum value. The peak at iteration 10 is the result of sampling noise. As desired, the optimization algorithm was able to absorb this disturbance and to get back on its track.

VI. Conclusions and Future Work

This paper presents AS-CDG, a fully automated and DUV-independent novel system for coverage-directed generation (CDG). The system first finds the relevant parameters that most influence the coverage of the target events. This is followed by a numeric optimization of noisy functions. To overcome the flat landscape caused by the lack of evidence for the uncovered target events, we use an approximated target of neighboring events for which evidence exists.

We implemented the proposed flow in a CDG tool suite, and it is currently used in the verification of many units of high-end processors. The results indicate that in many cases, the flow and the tool suite can hit previously uncovered events with minimal effort by the verification team.

The entire system operates outside the existing design and verification environment. This means that the flow can be used "as is" in any verification environment that utilizes parametrized test-templates. Moreover, applying the flow does not require any special skills from the verification team. This overcomes a major hurdle faced by many previous CDG tools.

While the flow proved to be useful in covering individual uncovered events or small sets of closely related events, there are several aspects that require work before it can be extended to a large number of uncovered events. Specifically, the number of simulations required to hit each uncovered event, which is reasonable for a single target event or a group of related events, may be too high when many uncovered events are involved. We are currently investigating methods that use machine learning techniques to reduce the number of simulations per event by using the same simulations for several target events.

References

[1] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. Springer, 2004.

[2] S. Ur and Y. Yadin, "Micro-architecture coverage directed generation of test programs," in *Proceedings of the 36th Design Automation Conference*, June 1999, pp. 175–180.

[3] R. Gal, E. Kermany, B. Saleh, A.Ziv, M. L. Behm, and B. G. Hickerson, "Template aware coverage: Taking coverage analysis to the next level," in *Proceedings of the 54th Design Automation Conference*, June 2017, pp. 36:1–36:6.

[4] A. Conn, K. Scheinberg, and L. Vicente, *Introduction to Derivative-Free Optimization*. Philadelphia: SIAM, 2009.

[5] R. Gal, E. Haber, B. Irwin, B. Saleh, and A. Ziv, "How to catch a lion in the desert: on the solution of the coverage directed generation (CDG) problem," *Optimization and Engineering*, 2020.

[6] C. Kelley, *Implicit Filtering*. Philadelphia: SIAM, 2011.

[7] P. Mishra and N. Dutt, "Automatic functional test program generation for pipelined processors using model checking," in *Seventh Annual IEEE International Workshop on High-Level Design Validation and Test*, October 2002, pp. 99–103.

[8] I. Wagner, V. Bertacco, and T. Austin, "Microprocessor verification via feedback-adjusted Markov models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 6, pp. 1126–1138, June 2007.

[9] S. Tasiran, F. Fallah, D. G. Chinnery, S. J. Weber, and K. Keutzer, "A functional validation technique: biased-random simulation guided by observability-based coverage," in *Proceedings of the 2001 International Conference on Computer Design*, September 2001, pp. 82–88.

[10] J. Smith, M. Bartley, and T. Fogarty, "Microprocessor design verification by two-phase evolution of variable length tests," in *Proceedings of the 1997 IEEE Conference on Evolutionary Computation*, 1997, pp. 453–458.

[11] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in *Proceedings of the 40th Design Automation Conference*, June 2003, pp. 286–291.

[12] S. Yang, R. Wille, D. Große, and R. Drechsler, "Coverage-driven stimuli generation," in *15th Euromicro Conference on Digital System Design*, 2012, pp. 525–528.

[13] F. Wang, H. Zhu, P. Popli, Y. Xiao, P. Bogdan, and S. Nazarian, "Accelerating coverage directed test generation for functional verification: A neural network-based framework," in *Proceedings of the Great Lakes Symposium on VLSI*, 2018, pp. 207–212.

[14] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: coverage-directed fuzz testing of RTL on fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–8.

[15] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in *Proceedings of the 40th Design Automation Conference*, 2003, pp. 286–291.

[16] R. Gal, H. Kermany, A. Ivrii, Z. Nevo, and A. Ziv, "Late breaking results: Friends - finding related interesting events via neighbor detection," in *Proceedings of the 57th Design Automation Conference*, July 2020.