Constraint Generation for Software-Based Post-Silicon Bug Masking with Scalable Resynthesis Technique for Constraint Optimization

Chia-Wei Chang[1], Hong-Zu Chou[2], Kai-Hui Chang[3], Jie-Hong Roland Jiang[2], Chien-Nan Jimmy Liu[1], Chiu-Han Hsiao[4] and Sy-Yen Kuo[2]

[1]Electrical Engineering Department, National Central University, Jhongli, Taiwan
[2]Electrical Engineering Department, National Taiwan University, Taipei, Taiwan
[3]Avery Design Systems, Inc., Andover, MA, USA
[4]Department of Information Management, National Taiwan University, Taipei, Taiwan
E-mail: chiuhan@nmi.iii.org.tw, sykuo@cc.ee.ntu.edu.tw

Abstract

Due to the dramatic increase in design complexity, verifying the functional correctness of a circuit is becoming more difficult. Therefore, bugs may escape all verification efforts and be detected after tape-out. While most existing solutions focus on fixing the problem on the hardware, in this work we propose a different methodology that tries to generate constraints which can be used to mask the bugs using software. This is achieved by utilizing formal reachability analysis to extract the conditions that can trigger the bugs. By synthesizing the bug conditions, we can derive input constraints for the software so that the hardware bugs will never be exposed. In addition, we observe that such constraints have special characteristics: they have small onset terms and flexible minterms. To facilitate the use of our methodology, we also propose a novel resynthesis technique to reduce the complexity of the constraints. In this way, software can be modified to run correctly on the buggy hardware, which can improve system quality without the high cost of respin.

Keywords

Bug repair, bug masking, logic synthesis

1. Introduction

Due to the dramatic increase in design complexity, verifying the functional correctness of a circuit is becoming more difficult. Therefore, bugs may escape all verification efforts and be detected after tape-out. One way to fix such post-silicon bugs is to change the layout and then do a respin. Although this approach is effective, it is also expensive due to the dramatic increase of respin costs. Another way is to integrate special logic into a circuit [17] so that bugs can be worked around. However, this method will not work if such logic has not been inserted in the first place. In addition, some bugs may not be able to be fixed this way if the capacity of the inserted logic cannot handle the bugs. Another approach that can work around post-silicon bugs is to change the firmware or software running on the hardware that can turn off certain features in the hardware. For example, Intel uses micro-instruction patches to fix bugs in their CPUs, and AMD suggested their users to turn off the translation look-aside buffer to fix a bug in the first-generation Phenom processor. Although such fixes more or less affect the circuit's performance, the circuit can still function correctly and does not need to be replaced. In this way, the cost to fix bugs can be significantly reduced. As a result, this approach is used frequently, especially in embedded systems whose software is controlled by the system company and will not be modified by end users. Since the functional correctness of a system is a key element of high quality designs, being able to fix bugs by repairing firmware of software can considerably improve design quality, especially after the system is shipped to customers and hardware replacement can be costly.

One major challenge to this software-based bug repair approach is to find a way to modify the software so that bugs can be masked without affecting system performance too much. For example, it is unwise to replace multiplication with a series of additions in a software if the multiplier is buggy for just a few combinations of numbers. However, finding the correct constraints for the software to mask hardware bugs can be highly challenging if the bug is non-trivial: manually analyzing the condition to trigger the bugs and deriving such constraints can be difficult. To address this problem, we propose a methodology that once a bug is found in the Register Transfer Level (RTL) code, we can analyze the conditions that trigger the bug and then produce constraints for the software team to modify their programs. This is achieved by performing formal code statement reachability analysis using symbolic simulation to identify all the symbolic conditions that can reach the buggy code, and then produce a Boolean function based on such conditions so that the bug will be exposed only if the function's output is one: this function represents the exact condition for the bug to appear. By synthesizing the function, a constraint, in the form of a single-output combinational circuit, can be generated and used by the software team to check their programs. Since such a constraint can be complicated, we also propose a novel resynthesis technique to simplify the constraint so that it can be handled efficiently. To this end, we observe that such constraints typically have small onset terms and the offset terms are flexible – bugs occur infrequently

and it is fine to have some false negatives. Based on this observation, we propose a resynthesis technique using Craig interpolation that is especially suitable for optimizing Boolean functions with small onset terms and flexible offset terms. Although there is trade-off between function complexity and the number of false negatives, our experimental results show that we can produce small constraints with minor increase in the number of false negatives in many cases. In addition, our resynthesis technique is much more scalable than traditional synthesis optimization techniques based on BDDs or implicant manipulations. This optimization undoubtedly will make our constraints easier to use in practice. Currently the software team needs to manually modify the software using the constraints. In our future work we will propose techniques to perform the modification automatically.

The rest of the paper is organized as follows. In Section 2 we provide necessary background. Our constraint generation methodology for software-based post-silicon bug repair is described in Section 3. Section 4 illustrates our resynthesis technique for constraint reduction. Empirical results are provided in Section 5 and Section 6 concludes this paper.

2. Background

In this section, we first provide an overview of post-silicon bug repair techniques, and then briefly describe formal reachability analysis that is utilized to extract bug conditions in our methodology. We also introduce Craig interpolation, constrained-random simulation, as well as several synthesis optimization techniques that utilize don't-cares.

2.1. Post-Silicon Bug Repair

One way to repair post-silicon bugs is to embed extra logic into a circuit so that patches can be applied to a buggy circuit. A semantic guardian [17] is an additional logic block that can prevent a design from entering undesirable states due to design bugs. In this methodology, users assign monitor points in a design and then simulate the circuit. The guardian generator considers the validated configurations as trusted states and automatically synthesizes the untrusted states as a guardian. By utilizing additional configurable registers, the guardian can be patched for future bug fixes. This approach can save a design from being recalled at the expense of small area increase and some performance degradation. Another related area is software fault-tolerance [15]. In this area, techniques such as N-version programming is capable of tolerating some transient or permanent hardware errors. However, these algorithms do not focus on design errors and require expensive resource overhead to develop multiple equivalent functions, which may not be economic. A more commonly-used approach to fix post-silicon bugs is to change the firmware or software running on the hardware. Fixing bugs directly is often much easier with this approach because it is easy to deploy software patches to end users. However, finding a correct fix without affecting system performance too much may not be easy.

In this work we provide a new methodology to facilitate this software-based bug repair process by producing constraints that will help designers identify the code segments which may expose hardware bugs.

2.2. Formal Reachability Analysis

Traditionally, reachability analysis is performed on state machines. In this work we use code-statement reachability analysis instead because it can target functional bugs in RTL code more effectively than working with design states. One major reason is that in this way, we can focus on the combinations of inputs that allow the buggy code to be executed instead of analyzing design states that may trigger the bugs. To this end, Chou *et al.* proposed a methodology to formally analyze code reachability using symbolic simulation [3]. Unlike logic simulation that only handles scalar values, symbolic simulation simulates symbols that represent both 0 and 1 simultaneously. Due to this exhaustive nature, symbolic simulation allows us to explore all possible paths to reach each code segment. In this way, we can obtain the symbolic condition when executing any line of the RTL code and use it to derive conditions that can trigger design bugs.

2.3. Craig Interpolation

Craig interpolation is a technique originated in mathematical logic [4]. Recently, it has become popular in verification and logic synthesis [2], [11], [12]. Craig interpolation theorem states that given a pair of Boolean formulas A and B, such that $A \wedge B = false$, there exists an intermediate formula I, such that $A \Rightarrow I$ and $I \Rightarrow \bar{B}$. I is called an *interpolant* and refers only to the common variables of formulas A and B. Intuitively, an interpolant I is an abstraction of A from the viewpoint of B.

An interpolant can be derived in linear time from the proof by resolution that $A \wedge B$ is unsatisfiable [11], [14]. There are several applications of Craig interpolation, such as model checking [11], functional dependency [12] and Boolean relation determination [7]. McMillan [11] utilized Craig interpolation to generate an approximated image operator that can be used in symbolic checking. Mishchenko *et al.* [12] proposed a logic optimization technique which uses interpolation to compute synthesis functional dependency. In [7], Jiang *et al.* addressed the scalability issue and demonstrated a method to extend determination capacity through interpolation.

2.4. Constrained-Random Simulation

Constrained-random simulation is widely used in simulation-base verification. It assigns random values to primary inputs and certain key variables so that legal input patterns can be generated quickly and easily. In this work we use constrained-random testbenches to model software behavior by assuming that the testbench can generate all possible program instructions that are allowed to be used by the software. This assumption is valid for many designs, such as embedded systems, where the program to be run is known in advance. We then derive constraints so that input sequences that will trigger the bugs will be detected and the software can be modified. Note that although the testbench can generate all possible inputs, it does not mean that all bugs

will be caught. For example, if the output checker was not comprehensive enough, incorrect outputs may not be flagged as errors. In addition, exhaustively verifying all possible inputs is time-consuming and may not be done in practice.

2.5. Synthesis Optimization Using Don't-Cares

In traditional two-level netlist optimization, one popular method to utilize don't-cares is to expand implicants into minterms that are don't-cares. Espresso [16] is a classic tool that implements many of the most commonly-used two-level techniques. For multi-level circuits, don't-cares have been utilized for rewiring [18] and node merging [13], [21]. To represent don't-cares more efficiently in multi-level circuits, Yamashita *et al.* proposed the concept of *Sets of Pairs of Functions to be Distinguished (SPFDs)* [18], [19]. To utilize don't-cares to optimize larger circuits, the SWEDE framework proposed by Chang *et al.* provides several methods that can scale better than traditional methods [2].

The above synthesis optimization techniques assume the care terms are fixed. In our application, however, only the onset terms are fixed and the offset terms are flexible. In other words, we can freely change offset terms into don't-cares for better optimization as long as the number of offset terms that become onsets does not increase too dramatically after synthesis. Such a synthesis problem can also be found in post-silicon validation and repair methods that try to use on-chip hardware to detect or work around bugs, such as the work by Wagner *et al.* [17] and Ko *et al.* [9]. These studies use techniques similar to those implemented in Espresso to solve this synthesis optimization problem, which are not scalable. In this work we propose a scalable method that can also be applied to solve the synthesis optimization problems used by these technique.

3. Constraint Generation for Software-Based Post-Silicon Bug Repair

The goal of this work is to automatically generate constraints for the software team to modify the program so that hardware bugs can be masked. In this section, we first formulate a synthesis problem that produces constraints for masking hardware bugs. We then describe how to utilize code-statement reachability analysis to produce the synthesis instance. Finally, we illustrate the overall flow of our methodology.

3.1. Problem Formulation

Given a circuit whose sequential depth is known as C that contains known bugs in its RTL code, as well as a constrained-random testbench that can generate all possible software sequence SEQ, our goal is to generate a constraint in the form of a single-output Boolean netlist such that if any software sequence $seq \in SEQ$ can trigger the bug, the output of the netlist becomes 1. By changing the software so that the output of the constraint is always 0, we can make sure the buggy code in the RTL will never be executed, and the hardware will work correctly. To achieve this goal, one can use the constraint as a software checker to determine whether or not the input sequence will cause the bug conditions to be true. This is carried out by iteratively checking the input sequence in intervals, potentially with register values at the first cycle of the interval, where interval length is the same as the sequential depth of the circuit. If the bug condition can be true, it indicates that certain input sequences will cause the hardware to execute buggy RTL code and produce incorrect outputs. By rewriting those input sequences so that the constraint's output becomes 0, the bug can be masked. Note that in this methodology, the generated constraints can be used by a software checker to check programs. They can also be used to modify a compiler so that the generated machine code can work around the hardware bug by construction. If the constraints are simple enough and programmable post-silicon bug-repair constructs are available, they can also be used on-line to flag the incoming inputs as problematic and switch the circuit to a safe mode to work-around the problem, as Wagner *et al.* suggests [17].

If the constraint is complicated, using such constraints can be inefficient. To address this problem, we also propose a novel resynthesis technique to reduce the complexity of the constraint, which will be described in detail in Section 4.

3.2. Constraint Generation Using Code-Statement Reachability Analysis

In order to extract all the conditions that cause the hardware to execute the buggy code, we utilize Chou's formal reachability analysis method [3] to explore all the paths to reach the buggy code statement, potentially under different conditions. This is achieved by replacing the random values in a constrained-random testbench with symbols and then perform symbolic simulation until the number of cycles reaches the required sequential depth. The procedure to extract bug conditions using reachability analysis is shown in Figure 1. In the procedure, we first perform reachability analysis on the RTL code. Whenever a buggy statement is executed, suppose it is executed under condition *sym_cond*, a SAT solver is used to check whether *sym_cond* is satisfiable or not. If *sym_cond* is satisfiable, it means the buggy statement can be reached, and *sym_cond* should be ORed with *bug_cond*. In this way, we can collect all the possible conditions that trigger the execution of the buggy statements using reachability analysis.

The *bug_cond* derived from formal reachability analysis is the condition for the hardware to work incorrectly. It is a Boolean function that when its output is 1, the inputs will trigger the bug. By synthesizing *bug_cond*, we will have a constraint that can be used to work around the hardware bug. Specifically, we use the constraint to check the correctness of the software. If any part of the software made the output of the circuit 1, then that part will expose bugs in the hardware and cannot be used.

3.3. Overall Methodology

Given a design's source code and a list of buggy code statements, our software-based bug repair methodology, shown in Figure 2, works as follows:

```
1  Perform formal reachability analysis on the buggy design;
2    whenever a buggy statement is executed
                    and the condition is sym_cond;
3      if (sym_cond can be 1)
4        bug_cond = bug_cond | sym_cond;
5  return bug_cond;
```

Figure 1: The procedure to extract bug conditions using reachability analysis.

1) Perform formal reachability analysis to extract all possible conditions which trigger the execution of buggy statements. This bug condition is then optimized to produce a Boolean netlist, which is called a constraint.
2) Execute the synthesis optimization described in Section 4 to reduce the complexity of the constraint.
3) Use the generated constraint to modify the software so that the original hardware bugs can be masked. This process is currently manual and will be automated in our future work.

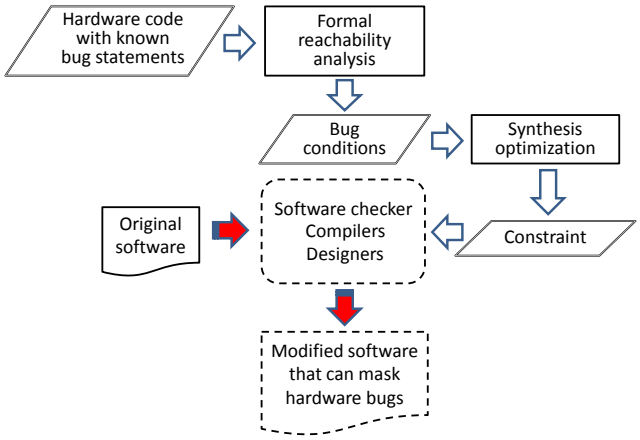

Figure 2: The overall flow of our software-based post-silicon bug repair methodology.

4. Constraint Optimization Via Resynthesis

Using the method described above, we can generate constraints to identify software segments that will expose hardware bugs. However, scalability still remains a concern. If the constraint is too complex, using it as a software checker can be inefficient. Therefore, it is desirable to reduce the complexity of the constraint.

The constraint produced by our methodology is essentially a Boolean function, or a single-output combinational circuit, that produces 1 on its output when bug conditions are detected. In this section we use "function" instead of "netlist" or "circuit" to refer to such constraints. However, they are essentially the same since it is easy to convert from one format to another. Such a function typically has small onsets. In addition, for the purpose of post-silicon bug repair, we observe that the function does not need to be exact: incorrectly marking some conditions as bug-triggering is acceptable, as long as the number of such false negatives is still small. In other words, we are allowed to change offset terms into onset terms as long as doing so can reduce function complexity, which in this work is measured using gate count after synthesizing the Boolean function. Based on this observation, we propose an optimization technique using Craig interpolation. In this section, we first formulate this optimization problem, and then describe how to use Craig interpolation to optimize a Boolean function whose offsets are flexible and has small onset terms. Finally, we perform an analytical study to provide more insights into our resynthesis technique.

4.1. Problem Formulation

We formulate our constraint optimization problem as a synthesis optimization problem as follows. A constraint is a Boolean function whose output is 1 when bug conditions are detected and 0 otherwise. All the onset terms are care terms because marking bug-triggering conditions as bug-free can cause the hardware to produce incorrect outputs, while all the offset terms are don't-cares because marking bug-free conditions as bug-triggering only reduces performance and is acceptable. The goal of our optimization is to reduce the complexity of the function, measured using gate count after synthesis, that does not make too many offset terms 1.

4.2. Function Optimization with Craig Interpolation

The first problem that needs to be solved in our optimization is to determine the offset care terms. As mentioned above, all the offset terms can be don't-cares. However, if we do not specify any offset terms as care terms, the synthesis tool can simply generate a constant 1 at the output, making the constraint useless. Therefore, it is obvious that we have to change certain offset terms into care terms when performing optimization. Intuitively, when more offset terms are turned into care terms, the bug will be more difficult to be hit. As a result, the number of false negatives that correct operations are determined as buggy operations will also reduce. However, the size of the resynthesized function will also increase. There is a trade-off between the complexity of the function and the number of false negatives.

In this work we randomly pick offset terms and make them care terms. As we will analyze in Section 4.3, although such an approach will stress on traditional resynthesis techniques due to the lack of logic structures in randomly selected terms, it seems to work well in our approach due to our use of Craig interpolation. Alternatively, if the input patterns that appear most frequently for the circuit are known, we can also make them care terms to make sure we will not mistakenly mark them as bug-triggering inputs. In this way, the impact on circuit performance can be reduced.

After offset terms are determined, we use Craig interpolation [2], [4] to produce an optimized netlist, and this technique works as follows.

1) Synthesize the truth table that represents the offset care terms to produce a single-output netlist: the output is 1 if the term is an offset care term, while all other terms

will produce 0 on the output of the netlist. We call this netlist $N0$.

2) Suppose the original netlist that represents the onset of the function is called $N1$, form a SAT problem by assuming that the onset and offset are true simultaneously.
3) Solve the SAT problem to produce a proof of unsatisfiability, and then derive the interpolant from the proof.
4) Create an AIG from the interpolant. The AIG is then optimized and technology mapped to produce a new netlist. Return the netlist as the optimized constraint.

4.3. Analysis of Our Method

Several traditional synthesis optimization techniques that can utilize don't-cares try to expand logic cubes one at a time. However, the number of cubes in the offset can grow exponentially as problem size increases. As a result, scalability becomes a major issue. Although one can improve the performance of such approaches by reducing the size of offset terms [10], it is still inefficient because the computation of offset terms can be challenging when most terms are don't-cares as in our case.

Unlike existing methods that rely on heuristic cube expansions, our approach randomly selects offset terms as care terms, which creates small offset bubbles in the input space. Craig interpolation is then used to quickly expand the offset bubbles. Figure 3 illustrates the concept of our synthesis optimization technique.

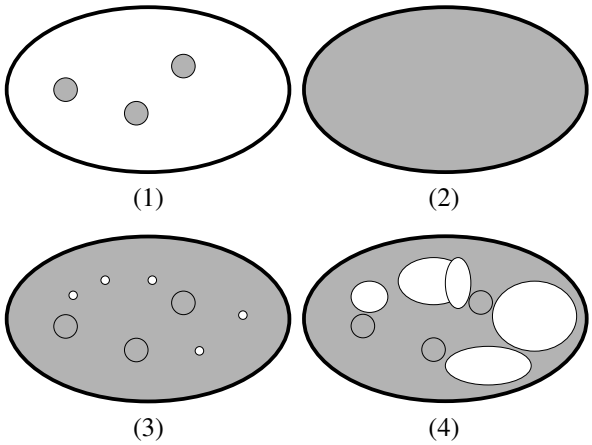

Figure 3: Illustration of our synthesis optimization technique. The oval represents the input space. Onset is denoted using light-gray and offset is white. (1) The original function, only a few inputs belong to the onset. (2) Since all offset terms are don't-cares, traditional techniques can generate a circuit that is constant 1, rendering the function useless. (3) In our work we randomly change certain offset terms into care terms, which creates small bubbles in the input space. (4) We use Craig interpolation to expand the offset bubbles so that the number of false negatives can be reduced.

Typically, such a random approach will stress on traditional synthesis methods because there is little relationship between the offset care terms. Since most synthesis algorithms work better when there are logic structures in the netlist being optimized, the lack of such structures will reduce the performance of those algorithms in terms of both runtime and resynthesis quality. On the other hand, Craig interpolation is based on the unsatisfiable core produced by solving the onset and offset netlists. Since the onset netlist is produced by symbolically simulating the design, it is highly structural and can provide valuable information for SAT solvers to derive useful conflict clauses. As a result, even though the offset terms lack logic structures, SAT solvers can still produce unsatisfiable cores that result in high-quality optimized functions. As our experimental results suggest, our optimization process is faster and more scalable than traditional approaches. In addition, it can produce small and accurate constraints most of the time.

Although Craig interpolation works well in our resynthesis approach most of the time, the fact that it is based on unsatisfiable cores also means the resynthesis quality can be unstable. Depending on the variable order that a solver performs SAT solving, the produced unsatisfiable cores can be quite different. This problem can be alleviated by running Craig interpolation several times using different random seeds and then pick the best resynthesized netlist. Alternatively, techniques that can minimize unsatisfiable cores are also being developed [5].

5. Experimental Results

We use the DLX design to evaluate our constraint generation method for software-based bug repair. DLX is a 32-bit processor with 5-stage pipeline [6]. The BugUnder Ground project from Michigan [23] provides a DLX implementation with 40 known bugs, where 34 of them are triggerable, and it is used in our experiments. Our goal is to produce a set of input constraints for working around the bugs. In our experiment, DLX was initialized to the state in which all registers were symbols. Since DLX has a 5-stage pipeline, all possible states can be reached in 7 cycles under this configuration. Therefore, symbolic simulation was executed for 7 cycles and the symbolic conditions for executing the buggy code at the 7th cycle were collected for constraint generation. Since the resynthesis quality using Craig interpolation can vary, we ran each configuration 5 times to obtain their average.

To extract all possible conditions that trigger the execution of buggy statements in the design's RTL code, we implemented the algorithm described in Figure 1 using a commercial symbolic simulator [22] that supports code reachability analysis. Our resynthesis algorithm is implemented using ABC based on its *inter* command [2]. To measure the complexity of the extracted bug conditions, we fed each Boolean function into ABC and then use its *resyn2* command to convert and optimize the function. Finally, we performed technology mapping based on the Cadence GSCLIB library to produce a mapped netlist. We then counted the number of cells in the mapped netlist to measure the complexity of the constraints. The results are shown in Table 1(a) under column "#Cells (Orig. Constr.)". From the results, we observe that the generated constraints are in the range of 100-250K cells, which

Table 1: Constraints generated using our method for each bug, 1000 offset care terms are used. (a) Complexity of the constraints generated for each bug, measured using the number of cells in the netlist. (b) Runtime of simulating 10000 random patterns on the generated constraints. It takes much shorter time to simulate the optimized constraints.

(a)

Case ID	#Cells (Orig. Constr.)	#Cells (Opt. Constr.)	Reduction ratio	False negative ratio
BUG1	106463	83172	21.88%	30.70%
BUG2	103346	65337	36.78%	22.96%
BUG3	108420	546	99.50%	0.85%
BUG4	110816	117727	-6.23%	0.02%
BUG5	138528	3595	97.40%	2.02%
BUG6	112910	167	99.85%	0.12%
BUG7	79466	98	99.88%	0.66%
BUG8	112066	17317	84.54%	10.20%
BUG9	175441	26676	84.80%	5.33%
BUG10	133494	7928	94,06%	4.36%
BUG11	62111	35924	42.17%	<0.01%
BUG12	62624	56386	9.96%	0.94%
BUG13	113347	4344	96.17%	3.41%
BUG14	122975	188238	-53.07%	28.55%
BUG15	105481	141639	-34.28%	<0.01%
BUG16	115183	22843	80.17%	4.5%
BUG17	113843	147735	-29.77%	27.24%
BUG18	125674	190432	-51.53%	<0.01%
BUG19	93252	15781	83.08%	19.27%
BUG21	117233	63428	45.90%	36.19%
BUG23	110892	100	99.91%	0.09%
BUG24	99497	6410	93.56%	1.81%
BUG25	112038	464	99.58%	1.19%
BUG26	110429	808	99.27%	1.86%
BUG27	98634	21038	78.67%	6.69%
BUG28	137268	50939	62.89%	11.56%
BUG30	95404	823	99.14%	2.02%
BUG32	98044	21834	77.73%	3.94%
BUG35	135844	293	99.78%	1.30%
BUG36	109570	650	99.41%	0.94%
BUG37	97081	26639	72.56%	4.31%
BUG38	112642	1043	99.07%	2.53%
BUG39	78148	63	99.92%	0.05%
BUG40	113180	39857	64.78%	14.36%

(b)

Case ID	Runtime (s) (Orig. Constr.)	Runtime (s) (Opt. Constr.)
BUG1	699.20	465.92
BUG2	672.18	117.29
BUG3	834.30	2.70
BUG4	1014.04	543.81
BUG5	1209.69	2.97
BUG6	1012.52	2.09
BUG7	549.00	1.97
BUG8	938.68	21.81
BUG9	1504.89	25.86
BUG10	1102.12	14.13
BUG11	410.80	68.50
BUG12	447.57	94.14
BUG13	928.91	5.87
BUG14	1316.29	971.71
BUG15	907.77	704.12
BUG16	1024.42	65.40
BUG17	954.63	785.59
BUG18	1129.17	1018.80
BUG19	738.08	18.79
BUG21	1071.39	267.75
BUG23	949.74	2.05
BUG24	785.88	9.98
BUG25	918.81	2.57
BUG26	975.78	2.76
BUG27	807.87	3.88
BUG28	1121.50	25.02
BUG30	739.26	3.08
BUG32	770.72	19.09
BUG35	1314.91	2.14
BUG36	951.17	3.00
BUG37	849.19	28.18
BUG38	984.91	2.97
BUG39	578.56	1.97
BUG40	937.67	133.35

may be too complex for software-based checkers, suggesting that constraint optimization is necessary.

To evaluate our constraint optimization technique, we performed logic simulation to generate 1000 random offset patterns and used them as offset care terms. We then used our resynthesis approach to optimize the constraint based on the new offset terms. The results are shown in Table 1(a) under column "#Cells (Opt. Constr.)", and the reduction ratio is also shown. The results show that our constraint optimization technique can reduce the complexity of the constraints most of the time: significantly smaller numbers of cells were used by the optimized constraints. However, five cases actually had larger cell count after optimization. The major reason is that the constraint size depends on the size of the extracted unsatisfiable core. Since the SAT engine that we use only tries to find one unsatisfiable core which is not necessarily the minimum one, the generated constraint could be larger after our optimization technique is applied. This problem can be alleviated using interpolant reduction techniques such as [1]. To measure how many false negatives were introduced for each optimized constraint, we ran 10000 random patterns on the original and the optimized constraints and measured the number of patterns that have their outputs changed to 1 due to the optimization. As the results show, there are 23 cases whose false negative ratios are smaller than 5%, and 11 of them have extremely small false negative ratios ($< 1\%$), suggesting that our constraint reduction approach can considerably reduce constraint complexity without creating too many false negatives.

Table 2: The effect of different number of offset care terms on the cell count of the optimized constraint. BUG2 is used as the case study.

#Offset care terms	#Cells (Opt. Constr.)	Reduction ratio	False negative ratio
1000	65337	36.78%	22.96%
1500	88483	14.38%	15.57%
2000	66771	35.39%	12.10%
2500	75394	27.05%	9.50%
3000	68945	33.29%	7.51%

To further evaluate the benefit of constraint optimization, we simulated 10000 random patterns using the original constraints

Table 3: Constraints optimized using our method, Espresso, and MVSIS.

			Our method			Espresso			MVSIS mfs		
Case ID	**#Onset terms**	**#Cells (Orig. Constr.)**	**#Cells (Opt. Constr.)**	**False negative ratio**	**Process time (s)**	**#Cells (Opt. Constr.)**	**False negative ratio**	**Process time (s)**	**#Cells (Opt. Constr.)**	**False negative ratio**	**Process time (s)**
c432	242460	15	76	4.48%	0.06	15	<0.01%	11184.55	15	<0.01%	6.85
c1355	73728	184	20	7.74%	0.14	1	12.42%	719.98	1002	<0.01%	9.74
c1908	34816	304	11	10.16%	0.15	4	23.45%	153.81	604	<0.01%	4.80
c6288	512	227	26	13.73%	0.15	6	35.90%	0.24	8	36.69%	0.13
s400	104173	37	24	3.78%	0.15	23	13.03%	1584.50	11	3.74%	1.64
s832	3824661	187	9	31.63%	0.14	5	14.85%	3 weeks	8	14.85%	65.62
s1494	2581	299	7	12.19%	0.14	2	12.22%	0.62	2	12.22%	0.14
s38584	8192	42	2	3.08%	0.12	3	9.35%	5.66	3	9.35%	0.18

and their optimized versions, and then compared their runtime. In this way, we can predict how much time will be saved by using the optimized constraints with a software-based checker. The results are presented in Table 1(b). From the Table, we observe that all the optimized constraints can be simulated much faster than the original ones, suggesting that our optimization can significantly reduce software verification time. It is also interesting to note that for the cases in which the optimized constraints have larger cell counts, such as BUG4, BUG14, BUG15, BUG17, and BUG18, their runtime is still smaller than original constraints. Although the reason is unclear, we suspect that Craig interpolation created netlist structures that are more suitable for logic simulation.

The number of offset care terms should affect the size of the optimized constraints and the number of false negatives. To measure this effect, we performed an experiment on BUG2 by varying the number of offset care terms that we used, and the results are shown in Table 2. The results show that as expected, the number of false negatives decreases with the increase in the number of offset care terms. Due to the instability of Craig interpolation, however, the trend is less obvious between cell-count reduction and the number of offset care terms. But in general, the trend still shows that when the number of offset care terms increases, cell-count reduction decreases. This result suggests that there is a trade-off between cell count and the number of false negatives.

To evaluate our constraint optimization method, we compared our algorithm with two public-domain tools: Espresso in SIS and the *mfs* command in MVSIS. Since Espresso and MVSIS cannot handle the complexity of DLX, we used ISCAS 85 and 89 benchmarks instead. However, ISCAS 89 benchmarks are sequential circuits; therefore, we used a commercial symbolic simulation tool [22] to simulate each benchmark for a few cycles to produce a combinational netlist. In addition, since we are focusing on resynthesizing netlists with small onset terms, we randomly chose some outputs from each benchmark and then ANDed them to produce an one-output netlist. By ANDing different outputs, we can make sure the produced netlist has small onset terms. The netlist is then optimized by ABC to produce a baseline netlist for comparison, and the gate count is shown in Table 3 under column "#Cells(Orig. Constr.)". We then applied logic simulation on the synthesized netlist to enumerate all onset terms and randomly selected 20 offset terms as care terms. The full onset terms, as well as the offset care terms, are then fed into SIS and MVSIS for optimization using "espresso" and "mfs" commands. The results are shown under columns "Espresso" and "MVSIS mfs". The resynthesis results of our method is shown under column "our method".

As the results shown in Table 3 suggest, Espresso has better cell-count reduction for most cases due to its exhaustive-search nature. However, for the case that has a huge number of onset terms such as s832, it took three weeks to produce the optimized netlist. On the other hand, mfs is based on heuristics and can better handle netlists with larger onset terms. However, the optimized netlist produced by mfs can be much worse than Espresso. Take c1335 for example, both optimized constraints have the same functionality, but their cell count are significantly different: the constraint produced by Espresso has only one cell, while the constraint produced by MVSIS mfs has 1002 cells. It is also observed that for Espresso and mfs, the process time used for optimization is dominated by the number of onset terms. The reason is that Espresso and mfs require all the care terms to be known. As a result, users have to enumerate all onset terms for optimization, which is very time-consuming and inefficient in practice. Compared with Espresso and mfs, our method achieves constraint optimization using an existing netlist that represents the onset terms and some randomly-selected offset care terms, making our process time much shorter than the other two methods. Furthermore, our method generated smaller netlists for many cases, showing that it can also produce high-quality results.

6. Conclusions and Future Work

In this paper we proposed a constraint generation methodology for software-based post-silicon bug repair. This methodology can automatically generate constraints for software so that hardware bugs can be worked around. Given a design that has known bugs in its RTL code and a constrained-random testbench that generates all possible input combinations of the software, we first perform formal code-statement reachability analysis to extract all possible conditions to execute the buggy code, and then produce a constraint in the form of a netlist by synthesizing and optimizing the bug condition. The constraint can then be used as a checker to verify whether a program can be executed on the hardware without problem. Since the

constraint may be complicated, we also propose a novel optimization method based on don't-cares that can dramatically reduce the size of the constraint. Our empirical results show that we can successfully generate constraints for the bugs in a DLX implementation, and our constraint optimization method can considerably reduce the complexity of the constraints. In our future work, we will utilize the constraints to repair the software automatically.

Acknowledgments

This research was supported by the Institute for Information Industry, Taiwan under Grant 99-FS-C02. This work was also partially supported by Republic of China (R.O.C.) National Science Council under Grants NSC 99-2221-E-008-104, 99-2923-E-002-005-MY3 and 99-2221-E-002-214-MY3. Their supports are greatly appreciated.

References

[1] John Backes and Marc Riedel, "Reduction of Interpolants for Logic Synthesis," *IWLS 2010*.

[2] K.-H. Chang, V. Bertacco, I. L. Markov, A. Mishchenko, "Logic Synthesis and Circuit Customization Using Extensive External Don't-Cares," *ACM TODAES*, Vol. 15(3), Article 26, May 2010.

[3] H.-Z. Chou, K.-H. Chang, and S.-Y. Kuo, "Optimizing Blocks in an SoC Using Symbolic Code-Statement Reachability Analysis," *ASPDAC 2010*, pp. 787-792.

[4] W. Craig, "Linear Reasoning: A New Form of the Herbrand-Gentzen Theorem," *Synthesis Logic*, Vol. 22(3), 1957, pp. 250-287.

[5] R. Gershman, M. Koifman and O. Strichman, "An Approach for Extracting a Small Unsatisfiable Core," *Form. Methods Syst. Des.*, 33(1-3):1-27, 2008.

[6] J. L. Hennessy, and D. J. Patterson, "Computer Architecture: A Quantitative Approach, 2nd edition," Morgan Kaufman, 1996.

[7] J. H. R. Jiang, H. P. Lin, and W. L. Hung, "Interpolating Functions from Large Boolean Relations," *ICCAD 2009*, pp. 779-784.

[8] H. F. Ko. and N. Nicolici, "On Automated Trigger Event Generation in Post-Silicon Validation," *DATE 2008*, pp 256-259.

[9] H. F. Ko. and N. Nicolici, "Resource-Efficient Programmable Trigger Units for Post-Silicon Validation," *ETS 2009*, pp. 17-22.

[10] A. Malik, R. K. Brayton, A.L. Sangiovanni-Vincentelli, "A Modified Approach to Two-Level Logic Minimization," *ICCAD 1988*, pp. 106-109.

[11] K. L. McMillan, "Interpolation and SAT-Based Model Checking," *CAV 2003*, pp. 1-13, LNCS 2725, Springer, 2003.

[12] A. Mishchenko, R. Brayton, J. H. R. Jiang, and S. Jang, "SAT-Based Logic Optimization and Resynthesis," *IWLS 2007*, pp. 358-364.

[13] S. M. Plaza, K.-H. Chang, I. L. Markov, and V. Bertacco, "Node Mergers in the Presence of Don't Cares," *ASPDAC 2007*, pp. 414-419.

[14] P. Pudlak, "Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations," *Journal on Symbolic Logic*, Vol. 62(3), 1997, pp. 981-998.

[15] D. A. Rennels, "Fault-Tolerant Computing" *Encyclopedia of Computer Science*, international Thomson Publishing – Europe, 1999.

[16] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization," *IEEE TCAD*, Sep. 1987, pp. 727-750.

[17] I. Wagner, V. Bertacco, "Engineering Trust with Semantic Guardians," *DATE 2007*, pp 743-748.

[18] S. Yamashita, H. Sawada, and A. Nagoya, "A New Method to Express Functional Permissibilities for LUT Based FPGAs and Its Applications," *ICCAD 1996*, pp. 254-261.

[19] S. Yamashita, H. Sawada and A. Nagoya, "SPFD: A New Method to Express Functional Flexibility," *IEEE TCAD*, Vol. 19(8), Aug. 2000, pp. 840-849.

[20] Y.-S. Yang, S. Sinha, A. Veneris and R. E. Brayton, "Automating Logic Rectification by Approximate SPFDs," *ASPDAC 2007*, pp. 402-407.

[21] Q. Zhu, N. Kitchen, A. Kuehlmann, A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't-Cares," *DAC 2006*, pp. 229-234.

[22] Avery Design Systems Inc., http://www.avery-design.com

[23] Bug UnderGround, http://bug.eecs.umich.edu

[24] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/abc.htm