Automated Methods for Eliminating X Bugs

Kai-Hui Chang, Yen-Ting Liu and Chris Browy
Avery Design Systems, Inc., Andover, MA, USA
changkh@avery-design.com, pheonix@avery-design.com.tw, cbrowy@avery-design.com

Abstract

Unknown values (Xs) may exist in a design due to uninitialized registers or blocks that are powered down. Due to limitations known as X-optimism and X-pessimism, such Xs cannot be handled correctly in logic simulation, producing inaccurate simulation values that can mask X bugs or corrupt simulation results. This can cause X bugs to escape verification and reduces design quality. To resolve such X-related issues, we propose a comprehensive methodology and several novel methods to detect masked Xs at the register transfer level and eliminate false Xs at the gate level. Our case studies show that the proposed methods are both effective and efficient.

Keywords

X verification, logic simulation, formal verification

1. Introduction

Physical synthesis optimizations and low-power requirements have left more registers uninitialized after reset compared with old design practices that reset most design registers. Additionally, power-down blocks during normal operation can create unknown values in design registers. In logic simulation, "Xs" are assigned to represent those unknown values which can be either 0 or 1 in real hardware. Such unknown values can result in non-deterministic states in a circuit, making the operation of the circuit unpredictable. Therefore, it is important to make sure those Xs do not propagate to important registers and corrupt circuit functionality.

Logic simulation is the most commonly-used verification method and is the main technique to catch X-related issues. However, logic simulation cannot handle Xs correctly due to X-optimism and X-pessimism problems shown in Figure 1. Typically, X-optimism problems are most serious in Register Transfer Level (RTL) simulation because they mask X bugs. For gate-level simulation, X-pessimism is the major issue because it can create numerous false Xs that corrupt simulation results and render logic simulation useless.

In this work we propose a comprehensive methodology that can detect bugs masked by X-optimism and eliminate false Xs generated from X-pessimism. Our methods leverage the proving power of formal methods and the scalability of logic simulation so that accurate X analysis can be efficiently performed. Our methodology contains the following components:

- *X-prescreener* analyzes test cases to select a small number of tests that should be analyzed to expose X bugs.

```
always @(posedge clk)
if (cond)
    b <= 0;
else
    b <= 1;
```

(a)

```
assign o = s & a | !s & b
```

When s is X, a is 1 and b is 1, o should be 1. However, logic simulation produces X.

(b)

Figure 1: (a) X-optimism example. If "cond" is X, else branch is executed and "b" becomes 1. But in hardware "b" can be either 0 or 1 depending on the real value of "cond". (b) X-pessimism example. Hardware value of "o" is 1 but logic simulation produces X.

- *XOPT Formal* formally identifies Xs masked due to X-optimism issues.
- *XOPT Sim* heuristically biases logic simulation toward taking conditional branches that are different from the default behavior to expose X bugs.
- *Safe Deposit Analysis* analyzes an input trace to identify Xs in registers that can be safely deposited with random values without masking any bugs.
- *SimXACT* eliminates all combinational false Xs so that gate-level simulation can work correctly and real X issues can be exposed.

Among the major components, XOPT Formal and SimXACT are based on our earlier work [1], [2], [3], [4], [5]. The overall X elimination methodology and the rest of the components are innovations first presented in this paper. The proposed methodology and techniques are currently in commercial production use, suggesting their effectiveness in handling X problems in industrial designs.

The rest of the paper is organized as follows. In Section 2 we review related and our previous work. Our methodology for eliminating X bugs is illustrated in Section 3, and Sections 4-6 describe each new component in detail. Case studies are provided in Section 7, and Section 8 concludes this paper.

2. Related and Previous Work

In this section we briefly review our previous work and related work that handle X issues.

2.1. Related Work

Most work that addresses the X problem in logic simulation focuses on the RTL because X-optimism in RTL simulation can easily mask bugs. The paper by Piper *et al.* [12] provided comprehensive background on how Xs are generated in industrial designs. They also surveyed several engineers to find

out the X problems they have and the solutions they need. However, their proposed solutions lack technical details and are only tested on a small OpenCores design, making their effectiveness on industrial designs unclear. Another solution from the industry is XProp from Synopsys [8]. In their solution, when an X is encountered at the condition of a statement, the assignments in all conditional branches are analyzed and their results are combined. If the value of a variable can differ under different branches, X is deposited to the variable. This technique creates additional Xs and produces simulation values closer to gate-level simulation. However, similar to gate-level simulation, false Xs may also be created, producing numerous false alarms that need to be analyzed.

Instead of fixing logic simulation, Haufe *et al.* [6] proposed to change the RTL coding style to reduce X-optimism so that RTL simulation results are closer to the gate level when Xs exist. However, this method can only alleviate X problems, not eliminate them. The X-Propagation solutions from Jasper [16] and Mentor [7] provide pure-formal methods to detect X problems. Similar to most formal methods, the user first defines input constraints and then specifies which registers to check. The formal engine then checks whether Xs in the specified registers can corrupt other registers. Such solutions can provide accurate proof that is valid for all cases. However, setting up formal analysis can be tedious, and scalability remains an issue.

To eliminate X-pessimism in gate-level simulation, the work by Nicholas [10] augments logic simulation to trace Xs and their negations and can eliminate false Xs at their convergence points. His technique is a simplified form of symbolic simulation and cannot efficiently trace the Xs when the design becomes too complicated. The solution from Petlin [11] can find X sources as well as the locations where such Xs can be trapped. However, he did not provide solutions on how to use the analysis to improve gate-level logic simulation accuracy. In addition, his method analyzes all possible paths for X-propagation, while our solution only analyzes the paths that caused the false Xs. The latter has significant performance advantage because the search space is much smaller. Salz *et al.* [13] also proposed a method for reducing X-pessimism in gate-level simulation. In their work, they assume a portion of the netlist, called a "sub-circuit", where X-pessimism can occur are given. For each sub-circuit, they make duplications of sub-circuit and use "X-splitters" to split Xs into 0 and 1. These concrete values are then fed into two different copies of the sub-circuit. The simulation results of the sub-circuits are then combined using "X-mergers". The problem with this approach is that identifying correct sub-circuits for analysis can be difficult. In addition, duplicating sub-circuits can require a lot of memory. The latter is especially problematic if more than one X can feed into the sub-circuit: the number of duplications can grow exponentially with each additional X source. Unlike Salz's work, our work identifies sub-circuits experiencing X-pessimism problems automatically and is much more efficient due to the use of formal solvers.

2.2. Our Previous Work

XOPT Formal is our solution to detect X-optimism problems in RTL code. It performs formal X-analysis by replaying the simulation trace from a test and looking for registers which have non-X values in logic simulation but are actually affected by Xs. It is described in detail in our earlier publications [2], [4], [5], and here is a brief summary of the flow:

1) Partition the design into smaller blocks.
2) For each block, partition the trace into several time intervals.
3) For each interval, replace Xs with symbols and perform symbolic simulation using the simulation values at primary inputs as stimulus.
4) At the end of the interval, for each register that has a non-X value in logic simulation, build a miter from its symbolic trace and check if the output of the miter can be 1. If so, the value of the register is affected by Xs and a masked X is found.

SimXACT is our solution for correcting X-pessimism problems in gate-level simulation. In [1] we presented how to prove whether an X is false and how to generate compact fixes to resolve X-pessimism issues. An overview of the proposed techniques is provided below:

1) Given an X at a register's D input, we build a Boolean function by tracing along the Xs until either a non-X value, a primary input, or a register output is encountered. We then use the *proveX* algorithm described in [1] to prove whether the X is false. This is achieved by using SAT to show if the Boolean function is a constant.
2) If the X is false, we use algorithms *ckt_minimize*1 and *ckt_minimize*2 to identify a small sub-circuit responsible for producing the false X. The former reduces the sub-circuit from the output to the inputs, while the latter goes the other way around.
3) After a small sub-circuit is identified, we generate behavior code to repair simulation. This is achieved by monitoring the values at the inputs of the sub-circuit and depositing the non-X value at its output.

In [2] we require the user to provide checkpoints and only check for false Xs at the checkpoints. In this work we propose two new methods, auto-fix and auto-monitor, that greatly simplify the use model. These enhancements and the new SimXACT flow will be described in Section 6.

3. Methodology for Eliminating Xs

In this section we introduce our methodology for eliminating X Bugs. For RTL, we focus on detecting Xs masked by X-optimism. For gate-level simulation, since there is typically no X-optimism at the gate level, our goal is to remove all false Xs so that real X bugs can be easily identified.

3.1. Detecting Masked X Bugs at the RTL

Even though X-optimism and X-pessimism problems can both arise in RTL simulation, X-pessimism is not a major issue because it does not mask bugs and designers can change

their RTL code to eliminate such false Xs. Therefore, for RTL simulation, our methodology focuses on identifying Xs masked by X-optimism. We call Xs that disappear in RTL simulation due to X-optimism *RTL X bugs*. In the following discussions, we call a branch that will be taken when the condition is X the "X-default branch". For "if" statement, the X-default branch is the "else" branch. For "case", it is the branch with "default" tag.

The flow to detect RTL X bugs is illustrated in Figure 2. In the flow, we optionally apply X-prescreener to select tests in a regression suite that should go through X analysis. The reason for this step is that performing X analysis takes time and effort. When the number of tests in a regression suite is large, it may be impractical to analyze all the tests. Therefore, it will be useful to select a few tests that can cover most X scenarios for further X analysis. The step can be skipped if the user already knows which tests should be analyzed or when the number of tests is small.

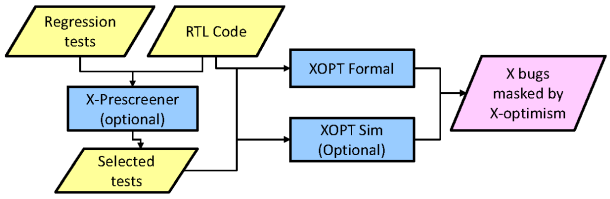

Figure 2: Methodology for detecting X bugs at the RTL.

After the tests are selected, we apply XOPT Formal that uses formal symbolic simulation to detect Xs masked by X-optimism in logic simulation. Additionally, we can apply XOPT Sim when running regression tests to create scenarios that are different from the X-default ones in order to expose X bugs. XOPT Sim intelligently deposits values to replace Xs so that simulation is biased toward taking the non X-default branch, which can change the execution paths of logic simulation and can potentially expose masked X bugs. XOPT Formal has been introduced in Section 2.2. X-prescreener and XOPT Sim will be described in detail in the next section.

3.2. Removing False Xs at the Gate Level

If a netlist is modeled using basic gate types such as AND, OR, XOR, INV, etc., then gate-level simulation has a special characteristic that simulating the combinational logic can only produce X-pessimism problems and cannot produce X-optimism problems. The reason is that for each gate, all its inputs will be evaluated during simulation, and the output of the gate is determined pessimistically — it produces a known value only if the Xs on the inputs are guaranteed not to propagate to the output. As a result, non-X values in gate-level simulation are always correct, while the Xs may be false. In our X-pessimism removal algorithm we utilize this characteristic to reduce formal analysis, which can provide considerable performance gain compared with pure formal methods. Since combinational cells in cell libraries are typically composed of such basic gates, most designs possess this characteristic.

Because there is no X-optimism in gate-level simulation, all X bugs will be exposed. However, real X bugs can be buried under numerous Xs produced due to X-pessimism. Therefore, for the gate level we focus on removing false Xs so that real X problems can be exposed. Our methodology for eliminating false Xs in gate-level simulation is shown in Figure 3. In the flow, we apply Safe Deposit Analysis (optional) to identify Xs that do not affect design operation — those Xs can be replaced with random values without masking any bugs. By reducing the number of initial Xs, fewer false Xs will be produced, thus reducing the effort for further X analysis. We then apply SimXACT that is guaranteed to eliminate all combinational false Xs. After false Xs are removed, all remaining Xs are real and should be inspected.

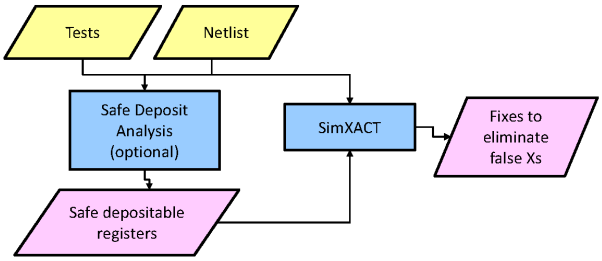

Figure 3: Methodology for generating fixes to eliminate false Xs in gate-level simulation.

4. RTL X Detection Methods

In this section we describe our new methods for detecting X problems at the RTL in detail.

4.1. X-prescreener

The purpose of X-prescreener is to select a small number of tests from the regression suite that can cover the most X scenarios. X-prescreener works as follows:

1) When an X is encountered in an if/case condition, create a node that records {file name, line number, hierarchical name to the instance}. Each node represents a unique X condition encountered during simulation.
2) After running a test, all the nodes created during the test will be saved and be associated with the test.
3) After all tests are analyzed, a maximum coverage problem is formed and solved to return a small number of tests that cover all the nodes.

Currently, we use a greedy algorithm to solve the maximum coverage problem by selecting the test that covers the most nodes at each step. Note that due to the difference in non-X values, the simulation paths for the same node may still be different for different tests. Therefore, it is not guaranteed that selected tests will cover all possible cases encountered in all tests. However, the selected tests should cover most X scenarios and provide a good starting point for tests to be analyzed if resource to run further X analysis is limited.

4.2. XOPT Sim

XOPT Sim is our innovation that biases logic simulation toward taking non X-default branches to expose RTL X bugs, and it works as follows.

1) Before a conditional statement such as "if" or "case" is executed, the condition is checked to see if it is X.
2) If the condition is not X, nothing is done. Otherwise, non-X values may be selectively deposited to variables involved in the condition, which produces a non-X value on the condition. Whether to deposit values and which values to deposit will be discussed later.
3) Since the condition has a non-X value, it may take a branch different from the X-default one, which changes simulation behavior and can potentially expose X bugs.

Take Figure 4 for example. When the valid clock edge arrives, suppose that variable *cond* is X, then "*cond* == 1" is also X. In this case, we may deposit a value to *cond*. Suppose that we deposit 1 to *cond*, then logic simulation will take the "if" branch and assign 1 to *a*. This changes the simulation result in that *b* is no longer 1. If the correct behavior requires *b* to be 1, then the bug will be exposed due to the change made by XOPT Sim.

```
always @(posedge clk)
if (cond == 1)
    a <= 1;
else
    b <= 1;
```

Figure 4: An example to show the effects of deposits made by XOPT Sim. If "cond" is X and is deposited with 1 before the condition is evaluated, then "if" branch will be taken instead of the X-default "else" branch.

Currently, for step (2), three modes are supported by XOPT Sim to determine whether to deposit values and which values to deposit. In the first mode, random values are always assigned to the variables involved in the condition so that the condition is no longer X. The condition may have any value, so logic simulation may or may not take a branch different from the X-default one. In the second mode, formal solving is used to solve for values so that a branch different from the X-default branch will always be taken. For "if" condition, the condition has to be 1. For "case", it can be any value that matches any condition tag that is not default. In the third mode, XOPT Sim randomly determines whether to deposit values. If values are to be deposited, formal solving is used to select the branch that is not X-default. This mode creates further randomization because Xs may be replaced later, allowing simulation to explore more varieties of scenarios.

Another problem that needs to be addressed is which variables should be the target of value deposits. Currently, two modes are supported. In the first mode, XOPT Sim deposits values to variables directly involved in the condition, no matter the variables are wires or registers. In the second mode, XOPT Sim deposits values only to registers or primary inputs. In other words, it traces the drivers of involved variables until it reaches registers or primary inputs for value deposits. Take Figure 5 for example, mode 1 will deposit a value to *cond*, while mode 2 traces the driver of *cond* and deposits a value to *d* because *d* is a register.

```
reg a, b, c, d;
wire cond;
assign cond = d;
always @(posedge clk)
if (cond == 1)
    a <= 1;
else
    b <= 1;
```

Figure 5: An example for illustrating different modes of deposit targets. The values may be deposited to variables directly involved in the condition only ("cond" in this example), or may be deposited only to registers ("d" in this example).

Typically, mode 2 is preferred because it replaces Xs in storage devices or primary inputs, which is closer to digital circuit behavior. However, mode 1 can still be useful because it can potentially create more varieties of execution paths, thus is more likely to expose design bugs. On the other hand, mode 1 may increase the number of false alarms because the values deposited at the wires may require conflicting values at the registers that can never be generated in real hardware.

XOPT Sim provides a heuristic for detecting design errors masked by X-optimism problems. Another commonly-used heuristic to expose RTL X bugs, called *random deposit*, is to replace Xs with randomly-selected 0/1 values and then perform logic simulation as usual. Compared with the latter method, XOPT Sim is more likely to find bugs because it allows Xs to propagate and can bias simulation toward choosing the branch that is not X-default, whereas random deposit is entirely based on probability. However, XOPT Sim is typically slower than random deposit because formal solving and additional code execution tracing have to be involved.

Compared with formal methods, XOPT Sim has advantages because it is based on logic simulation; therefore, it is more scalable than formal methods and there is no change in engineers' verification environments. As a result, it is much easier to set up and use than pure formal tools. However, XOPT Sim cannot provide comprehensive analysis like formal methods. To address this problem, XOPT Sim can be run with regression tests using different random seeds so that as many realistic operational scenarios can be explored as possible.

Some modern simulators provide the option for the user to change the X-default branches. Similar to XOPT Sim, this feature allows X-bugs to be heuristically exposed. However, changing X-default branches without replacing the Xs with concrete values can create too many false alarms because the bugs exposed may require conflicting value assignments to the Xs that are not possible in real hardware. By solving for concrete values to change simulation execution paths, XOPT Sim reduces the number of false alarms.

5. Safe Deposit Analysis

The purpose of Safe Deposit Analysis is to find Xs that do not affect any state registers — those Xs can be replaced by random values without masking any bugs. One common scenario to apply this analysis is to analyze the reset sequence to identify design Xs that will be eliminated after reset. By reducing the number of initial Xs, fewer X-pessimism problems will be created during simulation, thus reducing X analysis effort.

The Safe Deposit Analysis algorithm is shown in Figure 6. The inputs to the algorithm are the circuit (*ckt*) and an input trace (*trace*) to be analyzed. The algorithm returns registers whose Xs can be safely deposited because those Xs are eliminated during the simulation of the trace.

```
function safe_deposit_analyze(input ckt, input trace)
 1   foreach (reg ∈ ckt.get_x_regs())
 2     reg ← inject_symbol();
 3   for (n ← 0; n < trace.get_length(); n++)
 4     symbolic_simulate(trace.cycle(n));
 5     foreach (output ∈ ckt.get_outputs())
 6       straces ← straces ∪ output.strace;
 7   foreach (reg ∈ ckt.get_regs())
 8     straces ← straces ∪ reg.strace;
 9   foreach (strace ∈ straces)
10     if (check_x(strace))
11       unsafe_regs ← unsafe_regs ∪ strace.get_regs();
12   safe_regs ← ckt.get_x_regs() \ unsafe_regs;
13   return safe_regs;
```

Figure 6: Safe Deposit Analysis algorithm. Registers that can be safely deposited are returned.

The algorithm first injects a symbol into each register in the design that has an X value. Symbolic simulation is then performed to simulate the input trace in lines 3-4. At each cycle, we collect the symbolic traces at circuit outputs and save them to *straces* to capture Xs that will propagate out of the circuit. After simulating the input trace, we collect symbolic traces in design registers and add them to *straces* in lines 7-8 to capture Xs that still reside in the circuit — those Xs may affect design operations in the future. In lines 9-11, we use routine *check_x* to analyze each symbolic trace to see if it has a real X. This is achieved by building a miter to see if Xs can produce different values on the output of the symbolic trace [5]. If the X is real, we obtain the registers where the X came from using *strace.get_regs*() and add the returned registers to *unsafe_regs* because the Xs in those registers are not eliminated by simulating the input trace and cannot be safely deposited. In line 12 we subtract registers in *unsafe_regs* from all registers that have X values in the design and save the results to *safe_regs*. The registers in *safe_regs* have X values that will be eliminated by the input trace. Such Xs do not affect design operation and can be safely deposited. Finally, *safe_regs* is returned as the output of the algorithm.

6. SimXACT False X Elimination Method

In this section we present our SimXACT flow for correcting X-pessimism problems in gate-level simulation.

6.1. Overall Flow

The inputs to our X-pessimism elimination flow are a trace as input stimuli, a gate-level netlist, and a start time that the Xs should be checked for the first time to determine whether they are false or not. The output is auxiliary code that when the same X-pessimism conditions are encountered, those false Xs will be replaced with the correct values.

Our methodology works as follows:

1) At the start time we check Xs in register data inputs (typically denoted as "d") to determine if they are false.
2) For each false X, we trace the fan-in cone of the register input to find a portion of the cone, called a sub-circuit, whose inputs are real Xs and whose output is a false X.
3) We generate false X elimination solution and auxiliary code based on the sub-circuit to eliminate such Xs.
4) The generated fixes are applied by *auto-fix* to repair the current simulation values. *Auto-monitor* then finds further Xs that need to be checked throughout simulation.

After the analysis of the trace is finished, the generated auxiliary code can be used with future simulation to eliminate the false Xs. This flow allows gate-level simulation to produce correct results.

The first three steps in the flow were proposed in [1] and reviewed in Section 2. In the rest of the section we describe our innovations, auto-monitor and auto-fix, and then provide some discussions.

6.2. Auto-fix and Auto-monitor

Auto-fix is used in our flow to apply the generated fixes to repair simulation results instantly. Auto-fix works as follows: whenever a fix is generated during the SimXACT run, we begin monitoring the simulation values of the fix conditions and deposit the non-X value to replace the false X on the target variable when the conditions match. When the values no longer match the conditions, the force is released. This can be implemented via the programming interfaces of simulators.

At each cycle, our method is guaranteed to repair all false Xs because the number of false Xs is limited, and our procedure repairs at least one false X in each iteration. Therefore, the repair process will eventually end. Auto-fix then allows our methodology to find all the false Xs in one pass because fixing the false Xs at the current cycle can expose more false Xs at subsequent cycles. Since our X-analysis and repair methods can find and fix all the false Xs at any given cycle, with auto-fix we can guarantee that all the false Xs in all cycles within the trace will be repaired.

Auto-monitor examines simulation activities and applies X analysis to variables that can potentially have false Xs. More specifically, it monitors the gate-level simulation values on the fan-in cone of a register D input and checks for false Xs when the values in its fan-in cone change and the D input has an

X value. In other words, a variable is checked again if the variable has an X value and the variables in its fan-in cone have value changes.

Auto-monitor removes the burden to select time points to perform X analysis and greatly simplifies the application of our SimXACT flow. Since we check a variable again whenever it has fan-in changes, we can make sure no false Xs will be missed, thus ensuring the completeness of our analysis. With auto-fix and auto-monitor, if the starting time of X analysis is at the beginning of simulation, then the simulation is guaranteed to be free of combinational false Xs.

6.3. Discussions

To select the starting time of the analysis, one simple way is to start at the beginning of the simulation. This makes sure no false Xs will be missed. However, the number of Xs prior to reset can be large, making the analysis time-consuming. Since most pre-reset Xs typically do not affect simulation correctness, in practice we typically start SimXACT analysis a few cycles before reset is deasserted.

When applying our solutions to industrial designs, we found that false Xs that took a long time to prove typically do not cause gate-level simulation to fail verification. Further analysis shows that such false Xs are typically "accidental". For example, the carry bit of an adder can have a false X if some of its inputs happen to be 0 while the rest of the inputs are Xs. At the RTL, the output of the adder is most likely X and the X will not cause simulation problems. Since most verification environment is built around the RTL, the false X at the gate level becomes irrelevant. We utilize this observation to reduce the runtime of our analysis. For example, we ignore variables whose fan-in are all Xs or have more Xs than a user-defined number.

Even though SimXACT is efficient, analyzing a long trace for a large design can still be time-consuming. To accelerate SimXACT analysis on large designs, a partitioned flow can be deployed. In the flow, we partition the design so that only a part of the registers are analyzed in each partition. We then run all partitions in parallel to generate fixes. The fixes from all the partitions are then combined to produce a fix file. The fix file is then used for future simulation. Note that because each partition only generates partial fixes in each run, the process may need to be repeated to generate all the fixes.

7. Case Studies

The methodology and techniques described in this paper are currently in commercial production use. The effectiveness of XOPT Formal has been reported in [2], [4]. Since the work, XOPT Formal has successfully verified another 7 chips and found 6 masked X-optimism bugs. In this section we focus on providing more details for our new innovations, SimXACT and Safe Deposit Analysis, to demonstrate their effectiveness on eliminating false Xs in real industrial designs. For XOPT Sim, we found that mode 2 (deposit at registers) is almost always preferred because registers are where Xs reside in real hardware. However, due to programming interface support limitations in commercial logic simulators, tracing the correct RTL fan-in cone can be difficult. In addition, after values are deposited into registers, there is no guarantee that those values will propagate to the variables involved in the condition in time to bias logic simulation. Currently, we are able to bias simulation to take the non-default branch approximately 70% of the time. To make XOPT Sim more useful, additional support from logic simulators is required.

7.1. SimXACT

We integrated our solution with commercial simulators using the Verilog programming interface. Our formal engine is based on the ABC package from UC Berkeley [15]. All the benchmarks were done on high-end workstations typically found in simulation farms. Due to non-disclosure agreements with the companies, we cannot reveal detailed information of the designs and their testbenches, but all designs have multi-million gates and are real industrial circuits. The results of the case studies are summarized in Table 1.

Table 1: Summary of case studies.

Design	#Regs	Testbench	Simulator	Runtime	#Fixes
Case 1	142K	VMM-based	VCS	1h	450
Case 2	150K	VHDL	NC	30m	≈1000
Case 3	140K	Simple Verilog	VCS	10m	258
Case 4	3M	SystemVerilog	NC	3d	≈1000

The first case had approximately 142K registers. The testbench was VMM-based and the simulator was VCS. We analyzed a trace starting from a few cycles before reset deassertion until the first transaction is finished, which was approximately 15K cycles long. Runtime was an hour and we found 450 fixes. We found real X issues due to the inserted test structures that would otherwise have been masked by traditional random deposit methods. After addressing the real Xs, gate-level simulation was clean with the fixes.

The second case had approximately 150K registers. The testbench was in VHDL and the netlist was in Verilog. NC was used for simulating the design. We analyzed the reset sequence for the design using half an hour and simulation was clean with the fixes.

The third case had approximately 140K registers. The testbench was simple Verilog that generated a short hardware reset sequence which was 64 cycles long. We analyzed the trace using 10 minutes and found 258 fixes. Functional gate-level simulations using the fixes were all clean.

The fourth case had approximately 3M registers. We analyzed the trace for 3 days without partitioning and generated approximately 1K fixes. Gate-level simulation passed after applying the fixes.

In addition to these case studies, SimXACT has been applied to more than forty designs to fix X-pessimism issues in gate-level simulation. It also helped expose X bugs in at least five designs.

In general, according to our experience with the companies, setting up the environment typically takes only a few minutes and chip-level analysis can be done in a couple hours. The

found fixes generally repair X-pessimism problems and allow gate-level simulation to pass verification. For simulations that still fail, real X issues were always found.

Here we show an X bug that we found from a real design. The X was from a signal that indicates whether one calculation is done and further process can be performed. Before SimXACT was applied, random deposit was used, and tests passed without problem. With SimXACT, however, tests failed with Xs. Because false Xs are eliminated by SimXACT, by tracing X propagation in the waveform the source of the Xs was easily identified. The RTL code that generated the X is shown in Figure 7. In the design, cal1_done has reset, and cal2_start does not. In RTL simulation, when cal1_done is X, cal2_start is assigned 0, so it appears that cal2_start is working fine. However, in real hardware, at the reset cycle cal2_start can be either 0 or 1. For the cycle after reset, calculations based on cal2_start may or may not start based on the value of cal2_start, which can potentially produce incorrect results that corrupt the datapath. Since the probability for cal1_done to be 1 at the reset cycle is only $\frac{1}{64}$ in the design, random deposit did not expose the problem. But in real hardware the chip will be in a buggy state once every sixty-four times it is powered up, which is a serious problem. To address this issue, the designer added reset for cal2_start. In this case, SimXACT successfully prevented a respin.

```
always @(posedge clk)
if (cal1_done)
    cal2_start <= 1;
else
    cal2_start <= 0;
```

Figure 7: RTL code example that created X problems.

7.2. Safe Deposit Analysis

Safe Deposit Analysis is implemented using a commercial symbolic simulator called Insight [14]. We applied it to analyze the third case in the previous section and runtime was 2h12m. 4930 registers were identified to be safely depositable. We applied SimXACT after depositing those registers with random values and the number of fixes reduced from 258 to 156, suggesting that performing safe deposit analysis can indeed reduce the number of false Xs that will be generated during simulation.

8. Conclusion

In this work we proposed a comprehensive methodology and several innovations for finding X problems at the RTL and eliminating false Xs in gate-level simulation. Our first innovation, XOPT Sim, can heuristically expose X bugs in the RTL. Our second innovation, SimXACT, can eliminate all the combinational false Xs in gate-level simulation, allowing simulation to produce correct results when Xs exist. By eliminating false Xs, real X problems can be exposed and costly respin can be avoided. Our third innovation, Safe Deposit Analysis, can identify Xs that do not affect design operation and can be safely deposited with random values. By reducing the number of initial Xs, X analysis effort can be reduced. The methodology is in commercial production use, suggesting its effectiveness in solving X problems in industrial designs.

References

[1] K.-H. Chang and C. Browy, "Improving Gate-level Simulation Accuracy when Unknowns Exist", *DAC, 2012*, pp. 936-940.

[2] K.-H. Chang, H.-Z. Chou, H. Yu, D. Dobbyn and S.-Y. Kuo, "Handling Nondeterminism in Logic Simulation So That Your Waveform Can Be Trusted Again", *IEEE D&T*, DOI:10.1109/MDT.2011.75

[3] K.-H. Chang, Y.-T. Liu, C. Browy and C. Huang, "System and Method for Correcting Gate-level Simulation When Unknowns Exist", *United States Patent 8402405*, Mar 19, 2013

[4] H.-Z. Chou, H. Yu, K.-H. Chang, D.Dobbyn and S.-Y. Kuo, "Finding Reset Nondeterminism in RTL Designs – Scalable X-Analysis Methodology and Case Study", *DATE, 2010*, pp. 1494-1499.

[5] H. Z. Chou, K. H. Chang, and S. Y. Kuo, "Accurately Handle Don't-Care Conditions in High-Level Designs and Application for Reducing Initialized Registers," *IEEE TCAD*, Apr. 2010, pp. 646-651.

[6] C. Haufe and F. Rogin, "Ad-Hoc Translations to Close Verilog Semantics Gap," *workshop on DDECS, 2008*, pp. 1-6.

[7] K. Liu and R. Sabbagh, "Confidence in the Face of the Unknown: X-state Verification", *Verification Horizons*, Vol. 9, Issue 2, Jun. 2013

[8] G. Maturana, A. Salz and J. T. Buck, "Method and Apparatus for Simulating Behavioral Constructs Using Indeterminate Values", *US Patent 8271914*, Sep. 18, 2012

[9] A. Mishchenko, S. Chatterjee, R. Brayton, "DAG-Aware AIG Rewriting, A Fresh Look at Combinational Logic Synthesis", *DAC, 2006*, pp. 532-535.

[10] R. Nicholas, "System and Method for Improved Logic Simulation Using a Negative Unknown Boolean State", *US Patent 7761277*, Jul. 20, 2010

[11] O. A. Petlin, "Verification Systems and Methods", *US Patent Application 2010/0313175 A1*, Dec. 9, 2010

[12] L. Piper and V. Vimjam, "X-Propagation Woes: Masking Bugs at RTL and Unnecessary Debug at the Netlist", *DVCon, 2012*, session 5.3.

[13] A. Salz, G. R. Maturana, I.-H. Moon, L. R. Mcllwain, "Method and Apparatus for Reducing X-pessimism in Gate-level Simulation and Verification", *US Patent Application 2012/0072876 A1*, Mar. 22, 2012

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

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

[16] Jasper Design Automation Inc., http://www.jasper-da.com