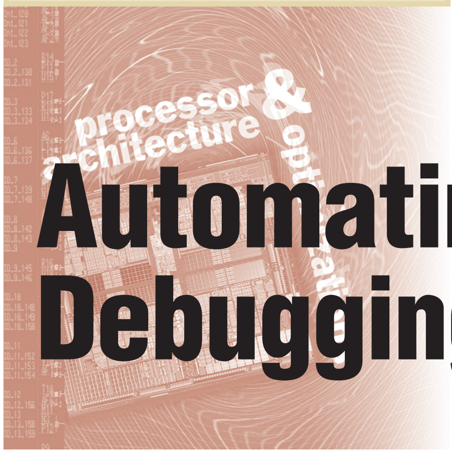


Automating Postsilicon Debugging and Repair

Kai-hui Chang, Igor L. Markov, and Valeria Bertacco
University of Michigan, Ann Arbor

Due to increasing semiconductor design complexity, more errors are escaping presilicon verification and being discovered only after manufacturing. As an alternative to traditional manual chip repair, the authors propose the FogClear methodology, which automates the postsilicon debugging process and thereby reduces IC development time and costs.

Due to the high complexity of modern semiconductor designs and increasing pressure to reduce their time to market, errors are more likely to escape verification and are often detected only after a chip has been manufactured. Postsilicon debugging has therefore become a crucial step in the design process, currently taking 35 percent of the time spent to complete a chip[1] and potentially consuming an even greater fraction in the future.

Given that the market window for many modern products is only a few years, the delay caused by two respins can dramatically reduce revenue or even kill the product. Carnegie Mellon University's Rob Rutenbar points out that postsilicon debugging can cost \$15 million to \$20 million and take six months to complete, yet few electronic design automation tools and algorithms address this problem.[2]

Postsilicon debugging is becoming more important because actual chips cannot be simulated presilicon with sufficient accuracy. Design deficiencies can involve complex thermal and inductive effects, while new semiconductor fabrication technologies can cause manufacturing glitches due to unexpected light-diffraction patterns and variability of material properties.

Nondeterministic defects are particularly difficult to work around because each affects only a small fraction of chips, but together they can significantly decrease yield and therefore increase cost. Consequently, a chip must be manufactured before developers can comprehensively validate it. In addition, silicon dies that undergo testing can operate at their intended frequency, which is orders of magnitude faster than functional and electrically accurate simulation.

PRESILICON VERSUS POSTSILICON DEBUGGING

Pre- and postsilicon debugging differ in four key ways.

First, design errors found before manufacturing include conceptual deficiencies that might not be fixable by automatic tools. In contrast, postsilicon functional bugs are often subtle errors that only affect the output responses of a few input vectors, and developers usually can implement fixes with very few gates, as the "Analysis of Presilicon and Postsilicon Bugs" sidebar explains. However, finding such fixes requires analyzing detailed layout information, making it a highly tedious and error-prone task.

Second, errors detected postsilicon typically include functional, electrical, manufacturing, and yield problems. However, issues identified presilicon are mostly limited to functional and timing errors. Problems that manage to evade presilicon validation are often difficult to simulate, analyze, or even duplicate. For example, Intel developed an entirely new methodology for postdiagnosis of electrical bugs that affect signal delays.[3]

Third, the observability of a silicon die's internal signals is extremely limited because most signals cannot be discerned directly.

Fourth, verifying the correctness of a fix is challenging because physically implementing a fix in a chip is difficult. So-called *metal fix* techniques, such as that described in the "Focused Ion Beam" sidebar, can alter

Analysis of Presilicon and Postsilicon Bugs

Semiconductor errors have many origins ranging from poor specifications to miscommunication among designers to plain designer mistakes. Table A lists the 15 most common error categories in microprocessor designs specified at the register-transfer level, as collected from seven student projects at the University of Michigan between 1996 and 1997.[1] Most students participating in this study are currently integrated circuit designers, therefore the bugs are representative of errors in industry designs.

As the table shows, most errors are simple and only require changing a few lines of code in a hardware description language, while complex and conceptual errors only contribute 7.9 percent of the total errors. This is not surprising for competent designers. Because even fewer such errors will escape presilicon verification, postsilicon functional bugs are often subtle and only affect the output responses to a few input vectors; their fixes can usually be implemented with very few gates.

Despite their subtlety, these faults occur more frequently with each design generation and can have serious consequences. As an example, a 2007 analysis of the Intel Core 2 processor by OpenBSD founder Theo de Raadt[2] identified 20-30 bugs that cannot be masked by BIOS or operating system updates. Some of these could be exploited by malicious software. De Raadt estimates that it would take Intel a year to repair these errors. It is particularly alarming that these bugs escaped Intel's verification and validation methodologies that are considered among the most advanced in the industry.

References

1. D.V. Campenhout, T. Mudge, and J.P. Hayes, "Collection and Analysis of Microprocessor Design Errors," *IEEE Design & Test,* vol. 17, no. 4, 2000, pp. 51-60.
2. T. de Raadt, "Intel Core 2," openbsd-misc list, Mailing List Archives, 27 June 2007; http://marc.info/?l=openbsd-misc&m=118296441702631.

Table A. Common microprocessor design error categories in seven student projects.

	Microprocessor project							
Error category	LC2	DLX1	DLX2	DLX3	X86	FPU	FXU	Average
Wrong signal source	27.3	31.4	25.7	46.2	32.8	23.5	25.7	30.4
Missing instance	0.0	28.6	20.0	23.1	14.8	5.9	15.9	15.5
Missing inversion	0.0	8.6	0.0	0.0	0.0	47.1	16.8	10.3
Timing and sophisticated, difficult-to-fix errors	9.1	8.6	0.0	7.7	6.6	11.8	4.4	6.9
Unconnected input(s)	0.0	8.6	14.3	7.7	8.2	5.9	0.9	6.5
Missing input(s)	9.1	8.6	5.7	7.7	11.5	0.0	0.0	6.1
Wrong gate/module type	13.6	0.0	11.4	0.0	9.8	0.0	0.0	5.0
Missing item/factor	9.1	2.9	5.7	0.0	0.0	0.0	4.4	3.2
Wrong constant	9.1	0.0	2.9	0.0	0.0	0.0	9.7	3.1
"Always" statement	9.1	0.0	2.9	0.0	0.0	0.0	2.7	2.1
Missing latch/flip-flop	0.0	0.0	0.0	0.0	4.9	5.9	0.9	1.7
Wrong bus width	4.5	0.0	0.0	0.0	0.0	0.0	7.1	1.7
Missing state	9.1	0.0	0.0	0.0	0.0	0.0	0.0	1.3
Conflicting outputs	0.0	0.0	0.0	0.0	7.7	0.0	0.0	1.1
Conceptual	0.0	0.0	2.9	0.0	3.3	0.0	0.9	1.0

the chip's metal layers, but these processes cannot create new transistors.

AUTOMATING POSTSILICON DEBUGGING

Due to these constraints, developers cannot apply most debugging techniques prevalent in early semiconductor design stages to postsilicon debugging. Consider, for example, the buggy layout shown in Figure 1a. A small modification in the layout that sizes up the driving gate requires changes in all transistor masks and refabrication of the chip, as Figure 1b illustrates, making the "simple" modification extremely expensive in postsilicon debugging.

Existing techniques for postsilicon debugging strive to provide more visibility and controllability for the silicon die.[1] Although such techniques greatly aid engineers, they do not automate the debugging process itself. To address this problem, we have developed a methodology that facilitates the automation of postsilicon debugging. Key innovations in our approach include support for postsilicon physical constraints and the ability to repair errors by subtle modifications of an existing layout. As Figure 1c shows, our techniques are aware of the physical constraints and can repair errors with minimal physical changes.

Focused Ion Beam

FIB is a technique that focuses a beam of gallium ions.[1] In the semiconductor industry, it can be used to modify an existing silicon die. For example, FIB can cut a wire or deposit conductive material to make a connection. However, it cannot create new transistors on a silicon die.

Ion milling can remove unwanted materials from a silicon die. When an accelerated ion hits the silicon die, the ion loses its energy by scattering the electrons and the lattice atoms. If the energy is higher than the atoms' binding energy, the atoms will sputter from the die's surface.

To complement material removal, *ion-induced deposition* adds new materials to a silicon die. This process directs a precursor gas, often an organometallic compound, to the surface of the die, which is then absorbed. When the incident ion beam hits the gas molecule, the molecule dissociates and leaves the metal constituent as a deposit. Similarly, FIB can deposit an insulator on the die. Because deposited materials can trap impurities such as gallium ions, the conductivity and resistivity of the deposited metal or insulator tend to be worse than those of the regular manufacturing process. However, this phenomenon does not pose serious challenges in postsilicon debugging because the changes are typically small.

FIB can either cut or reconnect top-level wires, but changing wires at lower levels is much more elaborate. Achieving this requires milling a large hole through the upper-level wires to expose the lower-level wire, which is then filled with oxide. A new smaller hole is then milled through the refilled oxide and metal is deposited down to the lower level. The affected upper-level wires might need to be reconnected in a similar way.

Reference

1. J. Melngailis, L.W. Swanson, and W. Thompson, "Focused Ion Beams in Semiconductor Manufacturing," *Wiley Encyclopedia of Electrical and Electronics Engineering,* John Wiley & Sons, 1999; http://mrw.interscience.wiley.com/emrw/9780471346081/eeee/article/W7020/current/abstract.

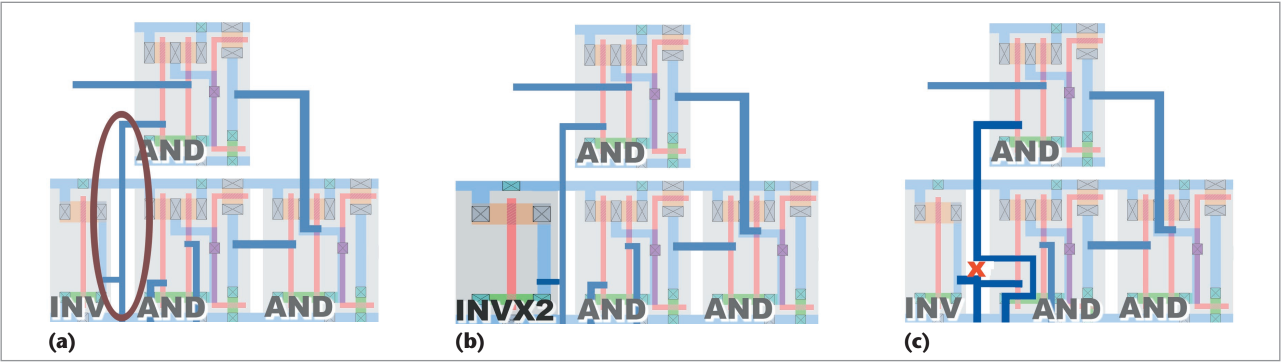


Figure 1. Postsilicon error repair example. (a) Original buggy layout with a weak driver (INV). (b) A traditional resynthesis technique finds a simple fix that sizes up the driving gate, but it requires expensive remanufacturing of the silicon die to change the transistors. (c) Our physically aware techniques find a more complex solution using symmetry-based rewiring, but it can be implemented with a metal fix and has a smaller physical impact.

To achieve these goals, we have developed algorithms to identify as many candidate bug fixes as practical, in terms of netlist and layout transformations. This is important in postsilicon debugging because often only a few transformations can satisfy all the physical constraints. On the other hand, we also exploit these constraints' highly restrictive nature to prune our algorithms' search space.

CURRENT POSTSILICON DEBUGGING METHODOLOGY

In his analysis of the major silicon failure mechanisms in microprocessors, Don Josephson reported that the most common failures, besides those due to dynamic logic, are drive strength (9 percent), logic errors (9 percent), race conditions (8 percent), unexpected capacitive coupling (7 percent), and drive fights (7 percent).[4] In addition, at the latest technology nodes, the *antenna effect*—electric charge accumulated in partially connected wire segments during the manufacturing process—can damage a circuit or reduce its reliability. Most of these issues must be addressed and resolved during postsilicon debugging. Our own work focuses on functional and electrical errors.

Figure 2 shows the current postsilicon debugging meth-

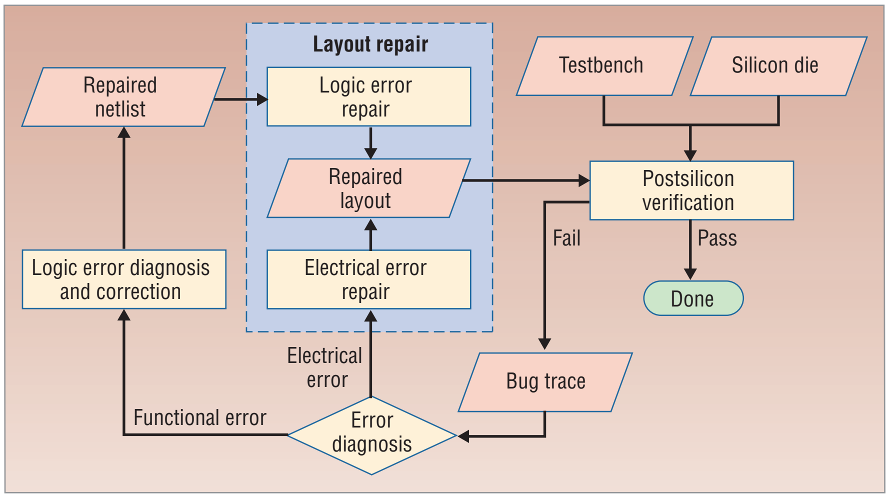


Figure 2. Current postsilicon debugging methodology. To validate a silicon die, engineers apply numerous test vectors to the die and then check their output responses. If the responses are correct for all the applied test vectors, the die passes validation. If not, the test vectors that expose the design errors become the bug trace, which engineers use to diagnose and correct the errors.

odology. To validate a silicon die, engineers apply numerous test vectors to the die and then check their output responses. If the responses are correct for all the applied test vectors, the die passes validation. If not, the test vectors that expose the design errors become the *bug trace*, which engineers use to diagnose and correct the errors.

After diagnosing the errors, engineers modify the layout to correct the errors and then validate the repaired layout again, continuing the process until the die passes verification. In contrast to presilicon verification, however, fixing all the errors as soon as they are diagnosed is often unnecessary in postsilicon debugging; in fact, repairing a fraction of them might be sufficient to enable further verification.

Functional errors

If engineers diagnose an error to be functional, they can resort to functional error repair techniques.[5] Even so, implementing fixes in the layout might not always be viable because these techniques do not take into consideration the physical aspects of the design. To find fixes compatible with a preexisting layout, engineers often generate several fixes, all equally valid from a functional standpoint, and then resort to tedious trial-and-error methodologies to select one that is compatible with the design's layout.

Electrical errors

Debugging electrical errors is often more challenging than debugging functional ones because engineers cannot use the available logic debugging tools. Although techniques to diagnose electrical errors exist (such as voltage-frequency shmoo plotting[6]), they are often heuristic in nature and require extensive expertise. Even if the errors' causes can be identified, finding valid fixes is still challenging because most existing resynthesis techniques require changes in logic cells and they are not amenable to metal fix.

To address this problem, researchers have recently developed techniques, such as *engineering change order* routing, that allow postsilicon metal fix.[7] However, ECO routing can only repair some of the electrical errors as it cannot reconnect wires and change the circuit's logic. Repairing more difficult bugs requires transformations that also utilize logic information. For example, one way to repair a drive-strength error is to exploit functional symmetries in the circuit as shown in Figure 1, and this can only be achieved by considering logic information.

FOGCLEAR METHODOLOGY

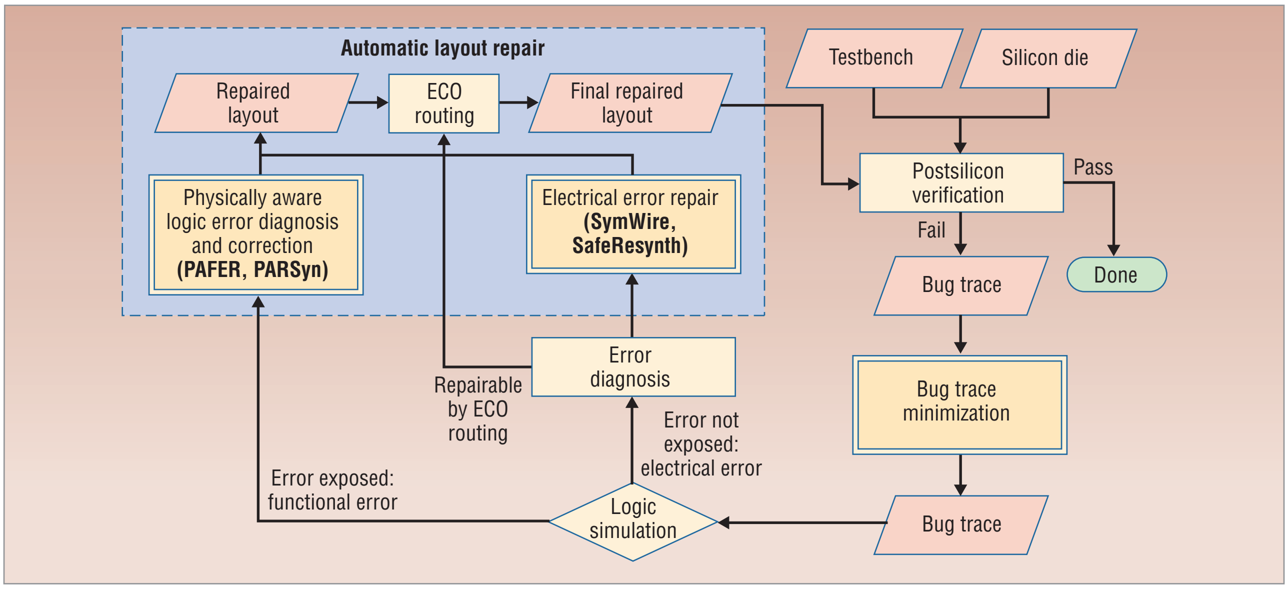
We have developed a postsilicon debugging methodology, called FogClear, that automates the manual effort currently required to isolate and fix bugs. Among its key elements are

- physically aware functional error repair (PAFER), which diagnoses and repairs functional errors with minimal perturbation to the layout; and
- physically aware resynthesis (PARSyn), which searches for netlist transformations that can be implemented with limited physical resources.

We also adapted earlier work on symmetry-based rewiring[8] and safe resynthesis[9] to search for layout transformations that can repair electrical errors.

In addition to postsilicon debugging, developers can apply FogClear to reduce the cost of respins. Because masks responsible for active device layers contribute most of the total mask cost, being able to reuse transistor masks greatly reduces respin cost, especially when it approaches $10 million per mask set at the 45-nm node. FogClear produces layout transformations that only involve changes in the metal layers and therefore allow transistor mask reuse. In addition, FogClear can accelerate the postsilicon debugging process and reduce the loss in revenue caused by delayed market entry.

Figure 3 shows the FogClear methodology. Because silicon dies offer execution speeds orders of magnitude faster than those provided by logic simulators, vendors rely on tests that can take several hours and sometimes



Automatic layout repair
Repaired layout
ECO routing
Final repaired layout
Testbench
Silicon die
Postsilicon verification
Pass
Fail
Done
Physically aware logic error diagnosis and correction (PAFER, PARSyn)
Electrical error repair (SymWire, SafeResynth)
Bug trace
Error diagnosis
Repairable by ECO routing
Bug trace minimization
Error not exposed: electrical error
Error exposed: functional error
Logic simulation
Bug trace

Figure 3. FogClear postsilicon debugging methodology. We first use bug trace minimization to reduce a trace's complexity. We then simulate the trace with a logic simulator to identify the error as functional or electrical. Next, we determine whether the error can be repaired by ECO routing tools or electrical error repair techniques. Finally, we route the repaired layout via ECO to produce the final layout, which we use to fix the silicon die for further verification.

days to execute on the silicon prototype. Consequently, when failing, these tests produce extremely long bug traces. To simplify error diagnosis, we introduce a step called *bug trace minimization*[10] that reduces the trace's complexity using several simulation-based methods. This approach is especially suitable for postsilicon debugging because simulation can be performed using the silicon die itself.

After simplifying a bug trace, we simulate the trace with a logic simulator using the netlist that produces the layout. If simulation exposes the error, then the error is functional, and PAFER generates a repaired layout; otherwise the error is electrical.

Currently, FogClear requires manual error diagnosis to determine the root cause of an electrical error. After identifying an electrical error, we determine whether ECO routing can repair the error. If so, we apply existing ECO routing tools;[7] otherwise, we deploy electrical error repair techniques based on reconnecting wires using functional symmetries (SymWire) or logic resynthesis (SafeResynth). We then route the repaired layout via ECO to produce the final layout, which we use to fix the silicon die for further verification.

PHYSICALLY AWARE FUNCTIONAL REPAIR

PAFER automatically diagnoses and fixes functional errors in the layout by modifying the combinational logic netlist of the design. To support the required change, it assumes that disconnected logic gates, or *spare cells*, are available. Various techniques have been proposed to insert spare cells in a layout.[11]

PAFER process

Given certain test vectors and their output responses, PAFER first uses simulation to generate a *signature* for each wire, a collection of the wire's simulated responses to the given test vectors.[12] Signatures provide an abstraction of the design because they are partial truth tables of the wires in the circuit.

Next, PAFER performs error diagnosis on the abstract model to identify the wires responsible for the errors and to suggest the correct function at one or more internal circuit nodes that would rectify the circuit's erroneous behavior.

PAFER then carries out error correction by resynthesizing the new logic functions from other signals in the circuit, after which it verifies that the new netlist is actually repaired. If this verification fails, PAFER uses the returned bug traces to extend and enrich the signatures and refine the abstraction. This process repeats until the new netlist passes verification.

PARSyn algorithm

To support the layout changes required in functional error repair, we developed the PARSyn resynthesis algorithm.

Resynthesis in postsilicon debugging is considerably different than traditional resynthesis because the number and type of spare cells available is often limited. Therefore, PARSyn's baseline design exhaustively tries all possible combinations of spare cells and input signals to find viable resynthesis netlists. Fortunately, the limitation in type of gates available makes it possible to prune the search space effectively.

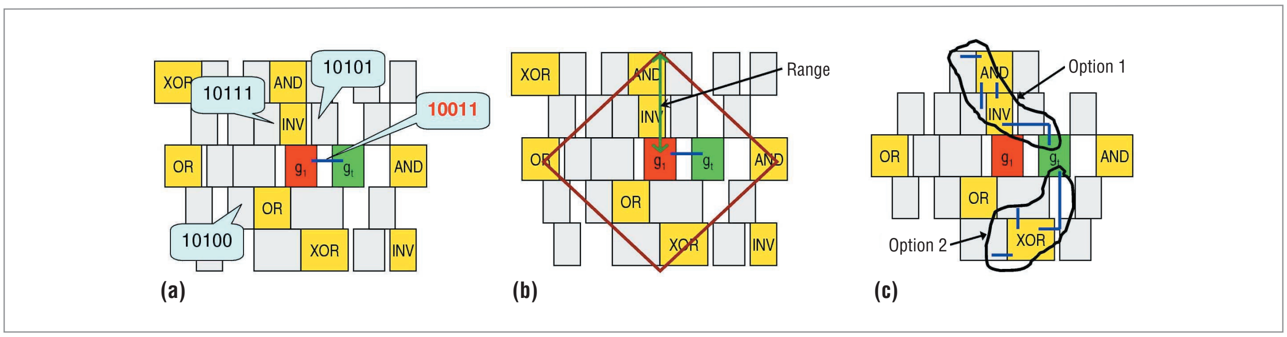


Figure 4. Functional error repair in an integrated circuit. (a) PAFER diagnoses that the wire driven by g_1 is erroneous and provides a corrected partial truth table using other cells as inputs. (b) PARSyn uses spare cells to perform Boolean manipulation of the signals, restricting its search to cells within a predicted range. (c) The algorithm generates various resynthesis netlists with different combinations of inputs and spare cells.

To further bound the search, PARSyn implements numerous logic search-pruning techniques,[5] including netlist connectivity analysis, to remove unpromising cells—for example, cells too far away from the erroneous wire—from the candidate pool. It also excludes cells in the erroneous wire's fanout cone to avoid generating combinational loops. Finally, PARSyn considers only spare cells within an engineer-selected distance of the erroneous wire's driver. One possible distance limit could be the maximum wirelength generated by a focused ion beam (FIB).

Functional error repair example

Figure 4 shows an execution example of functional error repair in an integrated circuit.

Figure 4a illustrates how PAFER diagnoses that the wire driven by g_1 is erroneous and provides a corrected partial truth table (signature). PARSyn's goal is to find a resynthesis netlist using other cells as inputs to generate the required truth table (partial truth tables for three cells are highlighted). PARSyn can then use spare cells, shown in yellow, to perform Boolean manipulation of the signals. As Figure 4b shows, PARSyn restricts the search to cells within the preselected range. It generates a number of resynthesis netlists with different combinations of inputs and spare cells, including the two options shown in Figure 4c. PARSyn will return only the resynthesis netlists that can be physically implemented and are functionally correct.[12]

AUTOMATING ELECTRICAL ERROR REPAIR

We have developed two techniques to alter erroneous wires and change their electrical characteristics without affecting the circuit's functional correctness.

SymWire rewiring technique

Symmetry-based rewiring uses symmetries to change the connections between gates. Figure 1c illustrates an example, where the inputs to the subcircuit composed of two AND and one OR gates are symmetric and therefore can be reconnected. The change in connections modifies the electrical characteristics of the affected wires and can be used to fix electrical errors. Because this rewiring technique does not perturb any cells, it is especially suitable for postsilicon debugging.

The SymWire rewiring algorithm first extracts various subcircuits from the original circuit, where the wire with the electrical error is one of the subcircuits' inputs. Each extracted subcircuit can contain one or more gates. The algorithm then detects symmetries in the inputs to the extracted subcircuits. If any of the symmetries involve the erroneous wire, SymWire can swap the input wires to repair the error.

Adapting SafeResynth to perform metal fix

Some electrical errors cannot be fixed simply by modifying a small number of wires, and a more aggressive technique is required. Because SafeResynth[9] can find alternative sources to generate the same signal using additional cells but without perturbing existing cells, we have adapted it to repair electrical errors as follows.

Assume that the error is caused by wire *w*, or by cell *g* driving *w*. We first use SafeResynth to find an alternative way to generate the same function at wire *w*. However, we only rely on the spare cells and need not insert new cells. Next, we disconnect *w* from *g* and use the new cells to drive *w*. Since *w* is now driven by other logic, we can change the electrical characteristics of both *g* and *w*. Note that SafeResynth subsumes cell relocation; therefore, it can also find layout transformations involving cell replacements.

CASE STUDIES

Our proposed techniques can repair drive-strength and coupling problems, as well as avoid the harm caused by the antenna effect. The following case studies serve as examples only; the same techniques can also be used to repair many other errors.

Insufficient driving strength

Drive-strength problems occur when a cell is too small to propagate its signal to all its fanouts within the designed timing budget. SafeResynth solves this problem by finding an alternative source to generate the same signal. As Figure 5a shows, the solution uses a new source to drive a fraction of the problematic cell's fanouts, reducing its required driving capability.

Coupling problems

Coupling between long parallel wires can delay signal transitions under some conditions as well as introduce unexpected signal noise. SafeResynth can prevent these undesirable phenomena by replacing the driver for one of the wires with an alternative signal source. Because the cell that generates the new signal will be at a different location, the wire topology can be changed. Alternatively, SymWire can also address the coupling problem: As Figure 5b shows, the affected wires no longer travel in parallel for long distances after rewiring, greatly reducing their coupling effects.

Antenna effect

Charge accumulated during semiconductor manufacturing in partially connected wire segments can damage and permanently disable transistors connected to such segments. Because the charge accumulated in a metal layer will be eliminated when the next layer is processed, it is possible to split the total charge with another layer by breaking a long wire and going up or down one layer through vias.

Manufacturers can alleviate occurrences of the antenna effect by intentionally inserting vias to route long wires on multiple layers. However, additional vias will increase the nets' resistance and slow down the signals. SymWire can find transformations that alter the metal layers assigned to several wires and reduce their antenna effects.

EMPIRICAL VALIDATION

To measure FogClear's effectiveness, we conducted two experiments. The first applied PAFER to repair functional errors in a layout, while the second used SymWire and SafeResynth to find potential electrical fixes. To facilitate metal fix, we preplaced spare cells uniformly in unused locations of the layouts, taking over about 70 percent of each layout's unused regions. These spare cells included INVERTERs, as well as two-input AND, OR, XOR, NAND, and NOR gates.

In applying PAFER, we set the search diameter parameter to 50 μm and limited resynthesis to generating netlists with at most two levels of logic per invocation. Under these conditions, only 45 spare cells are available for consideration, on average, when resynthesizing each signal. In our experimental findings, PAFER repaired more than 70 percent of the injected functional errors.[12] Repair failed when cells required to generate the target signals were too far away from the repair site to be considered. In such cases, metal fix is not a viable solution for bug fixing.

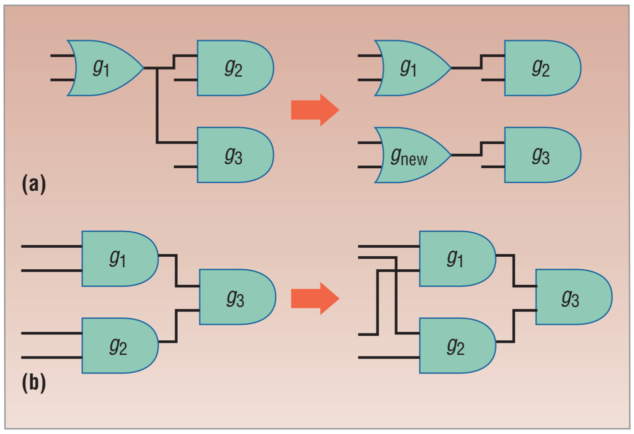


Figure 5. Case studies. (a) Cell g_1 has insufficient driving strength; SafeResynth uses a new cell, g_{new}, to drive a fraction of g_1's fanouts. (b) SymWire reduces coupling between parallel long wires by using symmetries to change their connections. This also changes metal layers and can alleviate the antenna effect.

With respect to electrical errors, both SymWire and SafeResynth altered more than half of the wires for most benchmarks, suggesting that they can effectively find layout transformations that change the erroneous wires' electrical characteristics. In addition, the number of affected metal segments was often small, indicating that both techniques have little physical impact on the chip; FIB can easily implement the layout modifications.

Due to increasing semiconductor design complexity, more errors are escaping presilicon verification and are discovered only later in prototype chips. While most steps in the integrated circuit design flow are highly automated, researchers have devoted little effort to the postsilicon debugging process, making it difficult and ad hoc.

Our proposed FogClear methodology, powered by novel techniques that enhance key steps in postsilicon debugging, systematically automates this process. The integration of logical, spatial, and electrical considerations in these techniques facilitates the generation of netlist and layout transformations to fix bugs, and it is complemented by sophisticated pruning techniques for more scalable processing.

Empirical results indicate that FogClear's key components—PAFER, PARSyn, SymWire, and SafeResynth—repair numerous functional and electrical errors in most benchmarks, demonstrating their effectiveness in postsilicon debugging. In addition, FogClear can reduce respin costs because the fixes it generates only affect metal layers. This accelerated postsilicon debugging process also enables a shorter respin cycle for the next prototype, thereby limiting revenue loss due to late-market entry.

References

1. M. Abramovici et al., "A Reconfigurable Design-for-Debug Infrastructure for SoCs," *Proc. 43rd Ann. Conf. Design Automation,* ACM Press, 2006, pp. 7-12.
2. R. Goering, "Post-Silicon Debugging Worth a Second Look," *EE Times,* 5 Feb. 2007; www.eetimes.com/showArticle.jhtml?articleID=197002823.
3. K. Killpack, C.V. Kashyap, and E. Chiprout, "Silicon Speedpath Measurement and Feedback into EDA Flows," *Proc. 44th Ann. Design Automation Conf.,* ACM Press, 2007, pp. 390-395.
4. D.D. Josephson, "The Manic Depression of Microprocessor Debug," *Proc. 2002 IEEE Int'l Test Conf.,* IEEE CS Press, 2002, pp. 657-663.
5. K-H. Chang, I.L. Markov, and V. Bertacco, "Fixing Design Errors with Counterexamples and Resynthesis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 1, 2008, pp. 184-188.
6. K. Baker and J.V. Beers, "Shmoo Plotting: The Black Art of IC Testing," *IEEE Design & Test of Computers,* vol. 14, no. 3, 1997, pp. 90-97.
7. H. Xiang et al., "An ECO Algorithm for Resolving OPC and Coupling Capacitance Violations," *Proc. 6th Int'l Conf. ASIC,* vol. 2, IEEE Press, 2005, pp. 873-876.
8. K-H. Chang, I.L. Markov, and V. Bertacco, "Post-Placement Rewiring and Rebuffering by Exhaustive Search for Functional Symmetries," *Proc. 2005 IEEE/ACM Int'l Conf. Computer-Aided Design,* IEEE CS Press, 2005, pp. 56-63.
9. K-H. Chang, I.L. Markov, and V. Bertacco, "Safe Delay Optimization for Physical Synthesis," *Proc. 2007 Asia and South Pacific Design Automation Conf.,* IEEE CS Press, 2007, pp. 628-633.
10. K-H. Chang, V. Bertacco, and I.L. Markov, "Simulation-Based Bug Trace Minimization with BMC-Based Refinement," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, no. 1, 2007, pp. 152-165.
11. K-H. Chang, I.L. Markov, and V. Bertacco, "Reap What You Sow: Spare Cells for Postsilicon Metal Fix," to appear in *Proc. 2008 Int'l Symp. Physical Design,* ACM Press, 2008.
12. K-H. Chang, I.L. Markov, and V. Bertacco, "Automating Post-Silicon Debugging and Repair," *Proc. 2007 IEEE/ACM Int'l Conf. Computer-Aided Design,* IEEE Press, 2007, pp. 91-98.

Kai-hui Chang *is a senior member of the technical staff at Avery Design Systems, a supplier of functional verification products based in Andover, Massachusetts. His research interests include verification, debugging, and logic synthesis. Chang received a PhD in computer science and engineering from the University of Michigan, Ann Arbor. He is a member of the IEEE and the ACM. Contact him at changkh@umich.edu.*

Igor L. Markov *is an associate professor of electrical engineering and computer science at the University of Michigan, Ann Arbor. His research interests include physical design and physical synthesis for VLSI, quantum information and computation, formal and semiformal verification of digital circuits and systems, combinatorial optimization, and search in artificial intelligence. Markov received a PhD in computer science from the University of California, Los Angeles. He is a member of the IEEE Computer Society and the American Mathematical Society, as well as a senior member of the IEEE and the ACM. Contact him at imarkov@umich.edu.*

Valeria Bertacco *is an assistant professor of electrical engineering and computer science at the University of Michigan, Ann Arbor. Her research interests are in the areas of formal and semiformal design verification, with emphasis on full design validation and digital system reliability. Bertacco received a PhD in electrical engineering from Stanford University. She is a member of the IEEE and the ACM. Contact her at valeria@umich.edu.*