Automating Post-Silicon Debugging and Repair

Kai-hui Chang, Igor L. Markov, Valeria Bertacco
EECS Department, University of Michigan, Ann Arbor, MI 48109-2121
{changkh, imarkov, valeria}@umich.edu

ABSTRACT

Modern IC designs have reached unparalleled levels of overall complexity, resulting in more and more bugs discovered post-silicon. However, few EDA tools can assist engineers in post-silicon debugging, since it requires a high level of sophistication. In this work we develop a methodology and new algorithms to automate this debugging process. Key innovations in our techniques include support for the unusual physical constraints of post-silicon debugging and ability to repair functional errors through subtle modifications of an existing layout. Our proposed post-silicon debugging methodology (FogClear) can also repair some electrical errors while preserving functional correctness. Thus, by automating this traditionally manual debugging process, our contributions promise to greatly reduce engineers' debugging effort. As our empirical results show, we can repair more than 70% of the representative circuits automatically.

1. INTRODUCTION

Due to the high complexity of modern designs and the increasing pressure to reduce their time-to-market, errors are more likely to escape verification and are only found after a chip has been manufactured. Needless to say, such errors must be fixed before the Integrated Circuits (ICs) can be shipped to customers, making post-silicon debugging a crucial step in the design process. To this end, a recent EE Times article quotes: "post-silicon debugging is a dirty little secret that can cost $15 to $20 million and take six months to complete" [14]. Indeed, post-silicon debugging has become one of the most time-consuming part, 35% on average, of the chip design cycle [2]. Therefore, it is surprising that only few EDA tools and algorithms address this problem [14].

Post-silicon debugging, however, is becoming more important because silicon ICs offer several advantages not available in pre-silicon. One reason is that manufacturing defects are becoming increasingly difficult to simulate, including those caused by antenna, thermal and inductive effects, as well as diffraction patterns. Non-deterministic effects, such as manufacturing variability, pose even greater challenges. As a result, comprehensive validation of a chip can only be performed after tape-out. In addition, silicon dies allow *at-speed testing*, which is orders of magnitude faster than logic simulation and astronomically faster than electrically-accurate simulation. If a sufficiently strong post-silicon debugging methodology is available, a part of the verification effort can be shifted to post-silicon, taking some pressure off the enormous simulation farms used by leading hardware vendors to validate their designs. Unfortunately, such a methodology is not yet available today.

Pre-silicon and post-silicon debugging differ in several significant ways. First, conceptual bugs that require deep understanding of the chip's functionality often appear in pre-silicon stage only, and such bugs may not be fixable by automatic tools. On the other hand, post-silicon functional bugs are often subtle errors that only affect the output responses of a few input vectors, and their fixes can usually be implemented with very few gates. However, finding such fixes requires the analysis of detailed layout information, making it a highly tedious and error-prone task. As we will show later, our work can automate this process. Second, errors found post-silicon typically include functional and electrical problems, as well as those related to manufacturability and yield. However, issues identified in pre-silicon are predominantly related to functional and timing errors.[1] Problems that manage to evade pre-silicon validation are often difficult to simulate, analyze and even duplicate. Third, the observability of the internal signals in a silicon die is extremely limited. Most internal signals cannot be directly observed, even in designs with *built-in scan chains* [5], which enable access to sequential elements. Fourth, verifying the correctness of a fix is challenging because it is difficult to physically implement a fix in a chip that has already been manufactured. Although techniques such as *Focused Ion Beam (FIB)* exist, they typically can only change metal layers of the chip and cannot create any new transistor (this process is often called *metal fix*).[2] Finally, it is especially important to minimize the size of each change in post-silicon debugging because smaller changes are easier to implement with good FIB techniques, and there is a smaller risk of unexpected side effects. Due to these unusual circumstances and constraints, most debugging techniques prevalent in early design stages cannot be applied to post-silicon debugging. In particular, conventional physical synthesis and Engineering Change Order (ECO) techniques affect too many cells or wire segments to be useful in post-silicon debugging. As illustrated in Figure 1(b), a small modification in the netlist that replaces a gate with another one requires changes in all transistor masks and refabrication of the chip. To this end, we observe that a recent technique called SafeResynth [10] only selects netlist modifications that require minimal physical changes. This philosophy is adopted in our work to handle the unusual constraints of post-silicon debugging.

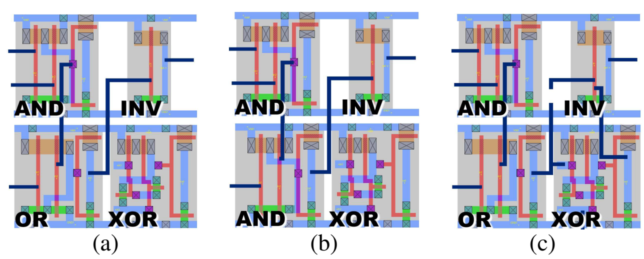

Figure 1: Post-silicon error-repair example. (a) The original buggy layout. (b) A traditional resynthesis technique finds a "simple" fix that only changes one cell type, but it requires expensive remanufacturing of the silicon die to change the transistors. (c) Our physically aware techniques find a more "complex" fix involving the use of a spare cell and several wire reconnections, but it can be implemented using only metal fixes and has smaller physical impact.

---

[1] Post-silicon timing violations are often caused by electrical problems and are only symptoms of such errors.

[2] Despite the impressive success of the FIB technique at recent fabrication technology nodes, the use of FIB is projected to become more problematic at future nodes, limiting the amount of allowable change and further complicating post-silicon debugging.

Existing techniques that address the post-silicon debugging problem strive to provide more visibility and controllability for the silicon die [2]. Although such techniques are great aids to engineers, they do not automate the debugging process itself. To address this problem, we propose new algorithms and a methodology that facilitate the automation of post-silicon debugging. These techniques can benefit from existing Design-For-Debugging (DFD) constructs but can also work well without them. Key innovations in our techniques include their support for the unusual physical constraints of post-silicon debugging and their ability to repair errors by subtle modifications of an existing layout. As illustrated in Figure 1(c), our techniques are aware of the physical constraints and can repair errors with minimal physical changes. To achieve these goals, our algorithms are exhaustive in nature in order to generate as many netlist and layout transformations as possible. This is important in post-silicon debugging because often only a few transformations can satisfy all the physical constraints. On the other hand, we also utilize these constraints in our algorithms because they can prune the search space effectively due to their highly restrictive nature. The main contributions of our work include: (1) a post-silicon debugging methodology, called *FogClear*, that automates the debugging process; (2) the *PARSyn* resynthesis algorithm that searches for netlist transformations which can be implemented with limited physical resources; (3) the *PAFER* framework that automatically diagnoses and repairs logic errors with minimal perturbation to the layout; and (4) the adaptation of symmetry-based rewiring [8] and *SafeResynth* [10] for post-silicon debugging to find layout transformations that can repair electrical errors. Empirical results show that our techniques are effective in repairing design errors and can greatly reduce engineers' debugging effort.

In addition to post-silicon debugging, FogClear can also be applied to reduce the cost of respins. As the data in [4] suggest, masks responsible for active device layers contribute about 68% of the total mask cost at the 100nm technology node. With mask costs approaching 10 million dollars per set at the 45 nm node (see Figure 2) [25], being able to reuse transistor masks greatly reduces the cost for a respin. This can be achieved using FogClear because the layout transformations it produces only involve changes in the metal layers and allow the reuse of the transistor masks.

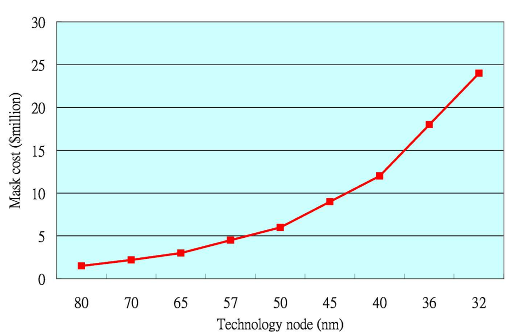

Figure 2: Estimated mask costs at different technology nodes [25]. The transformations produced by FogClear allow the reuse of transistor masks and thus greatly reduce respin costs.

The rest of the paper is organized as follows. In Section 2 we describe the current post-silicon debugging methodology and review some DFD techniques. The FogClear methodology that automates this debugging process is illustrated in Section 3. Our functional and electrical error repair techniques are explained in detail in Section 4 and Section 5, respectively. Experimental results are shown in Section 6, while Section 7 concludes this paper.

2. CURRENT POST-SILICON DEBUGGING METHODOLOGY

Josephson documented the major silicon failure mechanisms in microprocessors in [16], where the most common failures (excluding dynamic logic) are drive strength (9%), logic errors (9%), race conditions (8%), unexpected capacitive coupling (7%), and drive fights (7%). Another important problem at the latest technology nodes is the antenna effect, which can damage a circuit during its manufacturing or reduce the circuit's reliability. These problems often need to be solved via post-silicon debugging.

Figure 3 shows the current post-silicon debugging methodology. To verify the correctness of a silicon die, engineers apply a large number of test vectors to the die and then check their output responses. If the responses are correct for all the applied test vectors, then the die passes verification. If not, then the test vectors that expose the design errors become the *bug trace* that can be used to diagnose and correct the errors. The trace will then be diagnosed to identify the causes of the errors. Typically, there are three types of errors, including functional, electrical, and manufacturing/yield. In this work we only focus on the first two types of errors.

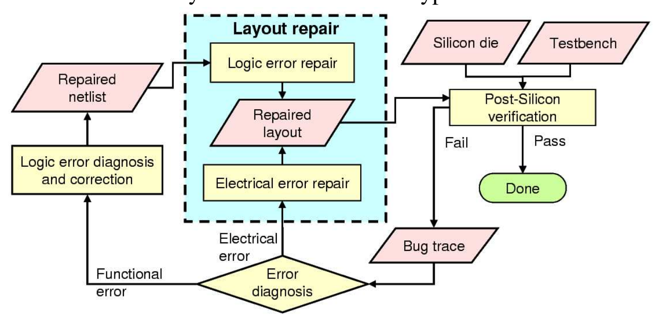

Figure 3: The current post-silicon debugging methodology. In this work we propose the FogClear methodology that automates this debugging process, which is shown in Figure 4.

After the errors are diagnosed, the layout will be modified to fix the errors, and the repaired layout will be verified again. This process keeps repeating until verification passes. In post-silicon debugging, however, it is often not necessary to fix all the errors because repairing a part of the errors may be enough to enable further verification. For example, a processor may contain a bug in its ALU and another bug in its branch predictor. If fixing the bug in the ALU already allows the die to be used in more testing, then it is not necessary to fix the branch predictor in the same die.

In the following subsections, we first describe two DFD techniques that can be used to facilitate post-silicon debugging. We then describe two important steps in the debugging methodology, including functional error repair and electrical error repair.

2.1 Design For Debugging

Without special constructs, only the values of a design's primary inputs and outputs can be observed in a chip, making its debugging extremely difficult. As a result, most modern designs incorporate a technique, called *scan test* [5], into their chips. This technique allows engineers to observe the values of internal registers and can greatly improve the design signals' observability.

In order to change the logic in a silicon die, spare cells are often scattered throughout a design to enable metal fix [17]. The number of spare cells depends on the methodology, as well as the expectation for respins and future steppings, and this number can reach 1% of all cells in mass-produced microprocessor designs. Alternatively, Lin *et al.* [18] proposed the use of programmable logic for this purpose. A recent start-up company [2, 26] provides a more comprehensive solution that further improves the observability of silicon dies and enables logic changes in the dies, and a success story can be found in [15]. In our work, we assume that scan test has been used, and spare cells are available for metal fix.

2.2 Functional Error Repair

If the errors are diagnosed to be functional, engineers can resort to the current logic error repair techniques, such as the work by Chang *et al.* [11], Veneris *et al.* [21], and Yang *et al.* [23]. These techniques can automatically diagnose design errors in combinational circuits and potentially find fixes to correct the errors. These fixes can then be used to repair the layout, usually via metal fix. However, implementing the fixes in the layout may not always be feasible because: (1) there may be insufficient spare cells to implement the resynthesis netlists; and (2) the wires to reconnect the cells may be too long to be generated by FIB. Although techniques that can generate various resynthesis netlists exist [24], they do not take physical information into consideration. To find fixes compatible with an existing layout, engineers often generate alternative fixes by perturbing their logic-level techniques and then resort to tedious trial-and-error methodologies. If no such fix can be found, engineers will have to do it manually. This is especially difficult because the netlists were automatically generated and have probably undergone many iterations of optimizations. As a result, it is difficult to understand the netlists even though the RTL code that produced them were designed by the engineers.

Our solution to this problem is discussed in Section 4, and it is based on the CoRé framework described in [11]. We adopted CoRé because: (1) it uses an abstraction-refinement scheme, which is more scalable than most existing techniques; (2) it only needs input vectors, output responses, and state values, which are easily available in post-silicon debugging; and (3) it provides a highly flexible interface that can adopt different resynthesis techniques. This is because CoRé operates on signatures, which are essentially partial truth tables of the nodes in the circuit. As a result, we can easily extend the framework to be physically aware by plugging in our new resynthesis technique.

The CoRé framework works as follows. Given certain test vectors and their output responses, it first uses simulation to generate signatures, which provide an abstraction of the design because signatures are partial truth tables of the wires in the circuit. Next, error diagnosis and resynthesis are performed on the abstract model to correct the errors. The repaired netlist is then verified. If verification fails, the returned bug traces are used to extend and enrich the signatures to refine the abstraction. This framework repeats until verification passes.

2.3 Electrical Error Repair

Debugging electrical errors is often more challenging than debugging functional errors because it does not allow the deployment of logic debugging tools that designers are familiar with. In addition, there are various reasons for electrical errors [16], and analyzing them requires profound design and physical knowledge. Although techniques to debug electrical errors exist (e.g., *voltage-frequency Shmoos* [3]), they are often heuristic in nature and require abundant expertise and experience. Even if the causes of the errors can be identified, finding valid fixes is challenging because most existing resynthesis techniques require changes in cells and do not allow metal fix. To address this problem, techniques that allow post-silicon metal fix have been developed recently, such as ECO routing [22]. However, ECO routing can only repair some of the electrical errors because it cannot find layout transformations involving logic changes. To repair more difficult bugs, transformations that also utilize logic information are required. For example, one way to repair a driving strength error is to identify alternative signal sources that also generate the same signal, and this can only be achieved when logic information is considered.

To this end, Chang *et al.* [9] proposed the concept of *physical safeness* to measure how well physical parameters are preserved by a physical synthesis technique. In their definition, techniques that do not perturb existing cells are physically safe; therefore, they can be used to repair electrical errors via metal fix. In light of this, we adapt their SafeResynth technique for post-silicon error repair. In addition, we develop a symmetry-based rewiring technique, called SymWire, that is physically safe and can repair electrical errors. Both techniques are able to find layout transformations involving netlist changes and are more powerful than ECO routing alone. These techniques are described in Section 5.

3. THE FOGCLEAR METHODOLOGY

Figure 4 shows our FogClear methodology that automates post-silicon debugging. When post-silicon verification fails, a bug trace will be produced. Since silicon dies offer simulation speeds orders of magnitude faster than those provided by logic simulators, constrained-random testing are used extensively, which can generate a bug trace that is extremely long. To simplify error diagnosis, we introduce a step called *bug trace minimization* in our methodology to reduce the complexity of the trace. To this end, we observe that many existing bug trace minimization techniques, such as the work by Safarpour *et al.* [20] or Pan *et al.* [19], rely heavily on SAT analysis and lack the scalability to handle these traces. On the other hand, the *Butramin* technique proposed by Chang *et al.* [7, 12] includes several simulation-based bug trace minimization methods, which are especially suitable for post-silicon debugging because simulation and bug trace minimization can be performed using the silicon die. As a result, in our FogClear methodology we develop the *SimButramin* component using the simulation-based methods described in [7, 12].

After the bug trace is simplified, we simulate the trace by a logic simulator using the netlist that produces the layout. If simulation exposes the error, then the error is functional, and PAFER is used to generate a repaired layout; otherwise the error is electrical. Currently, we still require manual error diagnosis to find the cause of an electrical error. After the cause of the error is identified, we check if the error can be repaired by ECO routing. If so, we apply existing ECO routing tools such as [22]; otherwise we use SymWire or SafeResynth to change the logic and wire connections around the error spot in order to fix the problem. The layout generated by SymWire or SafeResynth is then routed by an ECO router to produce the final repaired layout. This layout can be used to fix the silicon die for further verification.

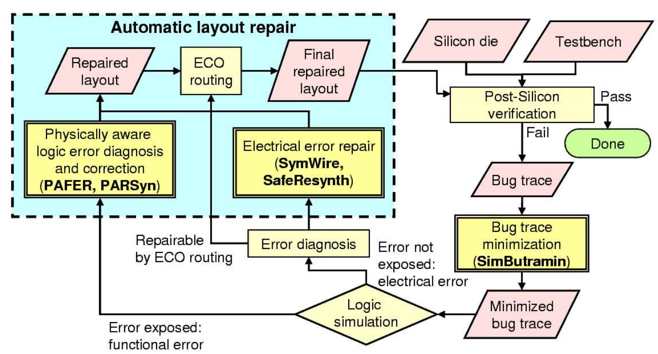

Figure 4: The FogClear post-silicon debugging methodology.

In the following sections, we will describe our functional and electrical error repair techniques in detail, including PAFER, SymWire and SafeResynth.

4. PHYSICALLY AWARE FUNCTIONAL ERROR REPAIR

In this section we describe our Physically Aware Functional Error Repair (PAFER) framework that automatically diagnoses and fixes logic errors in the layout by changing its combinational portion. In this context, we assume that state values are available, and we treat connections to the flip-flops as primary inputs and outputs. Our PAFER framework extends previous work in [11] which was empirically validated in the CoRé framework and shown to be scalable and flexible. To support the layout change required in logic error repair, we also describe a Physically Aware ReSynthesis (PARSyn) algorithm in this section.

4.1 The PAFER Framework

The algorithmic flow of our PAFER framework is outlined in Figure 5. Our enhancements to make the CoRé framework [11] physically aware are marked in boldface. Note that unlike CoRé, the circuits (ckt_{err}, ckt_{new}) in the PAFER framework now include layout information.

```
framework PAFER(ckt_err, vectors_p, vectors_e, ckt_new)
1   calculate ckt_err's initial signatures using vectors_p and vectors_e;
2   fixes ← diagnose(ckt_err, vectors_e);
3   foreach fix ∈ fixes
4     ckts_new ← PARSyn(fix, ckt_err);
5     if (every circuit in ckts_new violates physical constraints)
6       continue;
7     ckt_new ← the first circuit in ckts_new that does not violate
      physical constraints;
8     counterexample ← verify(ckt_new);
9     if (counterexample is empty)
10      return (ckt_new);
11    else
12      if (check(ckt_err, counterexample) fail)
13        fixes ← rediagnose(ckt_err, counterexample, fixes);
14    simulate counterexample and update ckt's signatures;
```

Figure 5: The algorithmic flow of the PAFER framework.

The inputs to the framework include the original circuit (ckt_{err}) and the test vectors ($vectors_p$, $vectors_e$). The output of the framework is a circuit (ckt_{new}) that passes verification and does not violate any physical constraints. In line 2 of the PAFER framework, the error is diagnosed, and the fixes are returned in $fixes$. Each fix contains one or more wires that are responsible for the circuit's erroneous behavior and should be resynthesized. In line 4 of the PAFER framework, *PARSyn* is used to generate a set of new resynthesized circuits (ckt_{new}), which will be described in the next subsection. These circuits are then checked to determine if any physical constraint is violated. For example, whether it is possible to implement the change using metal fix. In lines 5-6, that no circuit complies with the physical constraints means no valid implementation can be found for the current fix. As a result, the fix will be abandoned and the next fix will be tried. Otherwise, the first circuit that does not violate any physical constraints is selected in line 7, where the circuits in $ckts_{new}$ can be pre-sorted using important physical parameters such as timing, power consumption, or reliability. The functional correctness of this circuit is then verified as in the original CoRé framework. Please refer to [11, Section IV] for more details on this part of the framework.

4.2 The PARSyn Algorithm

The resynthesis problem in post-silicon debugging is considerably different from traditional ones because the numbers and types of spare cells are often limited. As a result, traditional resynthesis flow may not work because technology mapping the resynthesis function using the limited number of cells can be difficult. Even if the resynthesis function can be mapped, implementing the mapped netlist may still be infeasible due to other physical limitations. Therefore, it is desirable in post-silicon debugging that the resynthesis technique can generate as many resynthesis netlists as possible.

To support this requirement, our PARSyn algorithm exhaustively tries all possible combinations of spare cells and input signals in order to produce various resynthesis netlists. To reduce its search space, we also develop several pruning techniques based on logical and physical constraints. Although exhaustive in nature, our PARSyn algorithm is still practical because the numbers of spare cells and possible inputs to the resynthesis netlists are often small in post-silicon debugging, resulting in a significantly smaller search space than traditional resynthesis problems.

Our PARSyn algorithm is illustrated in Figure 6, which tries to resynthesize every wire ($wire_t$) in the given fix. In line 2 of the algorithm, *getSpareCell* searches for spare cells within *RANGE* and returns the results in *spareCells*, where *RANGE* is a distance parameter given by the engineer. This parameter limits the search of spare cells to those within *RANGE* starting from $wire_t$'s driver. One way to determine *RANGE* is to use the maximum length of a wire that FIB can produce. A subcircuit, ckt_{local}, is then extracted by *extractSubCkt* in line 3. This subcircuit contains the cells which generate the signals that are allowed to be used as new inputs for the resynthesis netlists. A set of resynthesis netlists ($resynNets_{new}$) is then generated by *exhaustiveSearch* in line 4. The cells in those netlists are then "placed" using spare cells in the layout to produce new circuits ($ckts_{new}$), which are returned in line 6.

```
function PARSyn(fix, ckt);
1   foreach wire_t ∈ fix
2     spareCells ← getSpareCell(wire_t, ckt, RANGE);
3     ckt_local ← extractSubCkt(wire_t, ckt, RANGE);
4     resynNets_new ← exhaustiveSearch(1, spareCells, ckt_local);
5     ckts_new ← placeResynNetlist(ckt, resynNets_new);
6   return (ckts_new);
```

Figure 6: The PARSyn algorithm.

To place the cells in a resynthesis netlist, we first sort spare cells according to their distances to $wire_t$'s driver. Next, we map each cell in the resynthesis netlist, the one closer to the netlist's output first, to the spare cell closest to $wire_t$'s driver. The reason behind this is that we assume the original driver is placed at a relatively good location. Since our resynthesis netlist will replace the original driver, we want to place the cell that generates the output signal of the resynthesis netlist as close to that location as possible. The rest of the cells in the resynthesis netlist are then placed using the spare cells around that cell.

The *exhaustiveSearch* function called in the PARSyn algorithm is given in Figure 7. This function exhaustively tries combinations of different cell types and input signals in order to generate resynthesis netlists. The inputs to the function include the current logic level ($logic$), available spare cells (*spareCells*), and a subcircuit (ckt_{local}) whose cells can be used to generate new inputs to the resynthesis netlists. The function returns valid resynthesis netlists in $netlists_{new}$.

In the function, $MAXLEVEL$ is the maximum depth of logic allowed to be used by the resynthesis netlists. So when $level$ equals to $MAXLEVEL$, no further search is allowed, and all the cells in ckt_{local} are returned (lines 1-2). In line 3, the search starts branching by trying every valid cell type, and the search is bounded if no spare cells are available for that cell type (lines 4-5). If a cell is available for resynthesis, it is deducted from the *spareCells* repository in line 6. In line 7 the algorithm recursively generates subnetlists for the next logic level, and the results are saved in $netlist_{sub}$. New netlists ($netlists_n$) for this logic level are then produced by

```
function exhaustiveSearch(level, spareCells, ckt_local)
1   if (level = MAXLEVEL)
2     return all cells in ckt_local;
3   foreach cellType ∈ validCellTypes
4     if (checkSpareCell(spareCells, cellType) failed)
5       continue;
6     spareCells[cellType].count- -;
7     netlists_sub ← exhaustiveSearch(level + 1, spareCells, ckt_local);
8     netlists_n ← generateNewCkts(cellType, netlists_sub);
9     netlists_n ← checkNetlist(netlists_n, spareCells);
10    netlists_new ← netlists_new ∪ netlists_n;
11  if (level = 1)
12    removeIncorrect(netlists_new);
13  return netlists_new;
```

Figure 7: The exhaustiveSearch function.

generateNewCkts. This function produces new netlists using a cell with type=*cellType* and inputs from combinations of sub-netlists from the next logic level. In line 9 *checkNetlist* checks all the netlists in $netlist_n$ and remove those that cannot be implemented using the available spare cells. All the netlists that can be implemented are then added to a set of netlists called $netlists_{new}$. If *level* is 1, the logic correctness of the netlists in $netlists_{new}$ is checked by *removeIncorrect*, and the netlists that cannot generate the correct resynthesis functions will be removed. The rest of the netlists will then be returned in line 13. Note that BUFFER should always be one of the valid cell types in order to generate resynthesis netlists whose logic levels are smaller than MAXLEVEL. The BUFFERs in a resynthesis netlist can be implemented by connecting their fanouts to their input wires without using any spare cells.

To bound the search in *exhaustiveSearch*, we implemented the logic pruning techniques described in Chang's GDS algorithm [11]. To further reduce the resynthesis runtime, we use netlist connectivity to remove part of the cells from our search pool: cells that are too many levels of logic away from the erroneous wire are removed. In addition, cells in the fanout cone of the erroneous wire are also removed to avoid the formation of combinational loops.

5. AUTOMATING ELECTRICAL ERROR REPAIR

The electrical errors found post-silicon are usually unlikely to happen in any given region of a circuit, but become statistically significant in large chips. To this end, a slight modification of the affected wires has a high probability to successfully repair the problem. However, being able to check this by accurate simulation and compare several alternative fixes increases the chances of a successful repair even further. In this section we first describe two techniques that can automatically find a variety of electrical error repair options, including *SymWire* and *SafeResynth*. These techniques are able to generate layout transformations that modify the erroneous wires without affecting the circuit's functional correctness. Next, we study three cases to show how our techniques can be used to repair electrical errors.

5.1 The SymWire Rewiring Technique

Symmetry-based rewiring changes the connections between gates using symmetries. An example is illustrated in Figure 9(b), where the inputs to cells g_1 and g_2 are symmetric and thus can be reconnected without changing the circuit's functionality. The change in connections modifies the electrical characteristics of the affected wires and can be used to fix electrical errors. Since this rewiring technique does not perturb any cell, it is especially suitable for post-silicon debugging.

In light of this, we propose an electrical error repair technique using symmetry-based rewiring, called *SymWire*, which is outlined in Figure 8. The input to the algorithm is the wire (w) that has electrical errors, and this algorithm changes the connections to the wire using symmetries. In line 1, we extract various sub-circuits (*subCircuits*) from the original circuit, where each sub-circuit has at least one input connecting to w. Currently, we extract sub-circuits composed of 1-7 cells in the fanout cone of w using breadth-first-search and depth-first-search. For each extracted sub-circuit, which is saved in (*ckt*), we detect as many symmetries as possible using function *symmetryDetect* (line 3). If any of the symmetries involve a permutation of w with another input, we swap the connections to change the electrical characteristics of w. In our implementation, we adopt the symmetry-detection technique described in [8] because their technique can detect a large number of symmetries and supports a variety of cell types.

```
Function SymWire(w)
1   extract subCircuits with w as one of the inputs;
2   foreach ckt ∈ subCircuits
3     sym ← symmetryDetect(ckt);
4     if (sym involves permutation of w with another input)
5       reconnect wires in ckt using sym;
```

Figure 8: The SymWire algorithm.

5.2 Adapting SafeResynth to Perform Metal Fix

Some electrical errors cannot be fixed by perturbing a few wires. For such errors, we need a more aggressive technique. We observe that the *SafeResynth* technique described in [10] can find alternative sources to generate a signal using an additional cell. Furthermore, their technique does not perturb existing cells. Therefore, we adapt SafeResynth to fix electrical errors, and it works as follows.

Assume that the error is caused by wire w or the cell g that drives w. We first use *SafeResynth* to find an alternative way to generate the same signal that drives w. In our work, however, we only rely on the so-called "space cells" that are embedded into the design but not connected to other cells. Therefore we do not need to insert new cells, which would be impossible to implement with metal fix. Next, we disconnect w from g and use the new cell to drive w. Since a different cell will be used to drive w, we can change the electrical characteristics of both g and w and potentially fix the error. Note that SafeResynth subsumes cell relocation; therefore, it can also find layout transformations involving replacements of cells.

5.3 Case Studies

In this subsection we show how our techniques can repair drive strength and coupling problems, as well as avoid the harm caused by the antenna effect. Note that these case studies only serve as examples, and our techniques can also be applied to repair many other errors.

Drive strength problems occur when a cell has insufficient driving capability to propagate its signal to all the fanouts within the designed timing budget. Our SafeResynth technique solves this problem by finding an alternative source to generate the same signal. The new source can then be used to drive a part of the fanouts of the problematic cell, thus reducing its required driving capability. An illustration of this process is given in Figure 9(a).

Coupling between long parallel wires that are next to each other can result in delayed signal transitions under some conditions and also introduces unexpected signal noise. Our SafeResynth technique can prevent these undesirable phenomena by replacing the driver for one of the wires with an alternative signal source. Since the cell that generates the new signal will be at a different location, the wire topology can be changed. Alternatively, SymWire can also

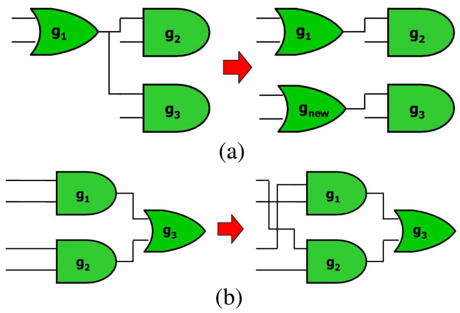

Figure 9: Case studies. (a) g_1 has insufficient driving strength, and SafeResynth uses a new cell, g_{new}, to drive a part of g_1's fanouts. (b) SymWire reduces coupling between parallel long wires by changing their connections using symmetries, which also changes metal layers and can alleviate the antenna effect.

be used to solve the coupling problem. As shown in Figure 9(b), the affected wires no longer travel in parallel for long distances after rewiring, which can greatly reduce their coupling effects.

Antenna effects are caused by the charge accumulated during semiconductor manufacturing in partially-connected wire segments. This charge can damage and permanently disable transistors connected to such wire segments. In less severe situations, it changes the transistor's behavior gradually and reduces the reliability of the circuit. 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. Based on this observation, *metal jumpers* [13] have been used to alleviate the antenna effect, where vias are intentionally inserted to change layers for long wires. However, the new vias will increase the resistivity of the nets and slow down the signals. To this end, our SymWire technique can find transformations that change the metal layers of several wires to reduce their antenna effects. In addition, it allows simultaneous optimization of other parameters, such as the coupling between wires, as shown in Figure 9(b).

6. EXPERIMENTAL RESULTS

To measure the effectiveness of the components in our FogClear methodology, we conducted two experiments. In the first experiment we apply PAFER to repair functional errors in a layout; while the second experiment evaluates the effectiveness of SymWire and SafeResynth in finding potential electrical fixes. To allow metal fix, we pre-placed spare cells uniformly using the whitespace in the layouts, and they occupied about 70% of each layout's whitespace. The types of the spare cells included: INVERTER, AND, OR, XOR, NAND, and NOR (all cells except INVERTER have two inputs). In the PAFER framework, we set *RANGE* to 50μm and $MAXLEVEL$ to 2. All the experiments were conducted on an AMD Opteron 880 workstation running Linux. The benchmarks were selected from OpenCores [27] except DLX, Alpha, and EXU_ECL. DLX and Alpha were internally developed benchmarks, while EXU_ECL was the control unit of OpenSparc's EXU block [28]. Our benchmarks are representative because they cover various categories of modern circuits, and their characteristics are summarized in Table 1. In the table, "#FFs" is the number of flip-flops and "#Cells" is the cell count of each benchmark. To produce the layouts for our experiments, we first synthesized the RTL designs with Cadence RTL Compiler 4.10 using a cell library based on the 18μm technology node. We then placed the synthesized netlists with Capo 10.2 [6] and routed them with Cadence NanoRoute 4.10.

Benchmark	Description	#FFs	#Cells
Stepper	Stepper Motor Drive	25	226
SASC	Simple Asynchronous Serial Controller	117	549
EXU_ECL	OpenSparc EXU control unit	351	1460
Pre_norm	Part of FPU	71	1877
MiniRISC	MiniRISC full chip	887	6402
AC97_ctrl	WISHBONE AC 97 Controller	2199	11855
USB_funct	USB function core	1746	12808
MD5	MD5 full chip	910	13311
DLX	5-stage pipeline CPU running MIPS-Lite ISA	2062	14725
PCI_bridge32	PCI	3359	16816
AES_core	AES Cipher	530	20795
WB_conmax	WISHBONE Conmax IP Core	770	29034
Alpha	5-stage pipeline CPU running Alpha ISA	2917	38299
Ethernet	Ethernet IP core	10544	46771
DES_perf	DES core	8808	98341

Table 1: Characteristics of benchmarks.

6.1 Functional Error Repair

To evaluate our PAFER framework, we chose several benchmarks and injected functional errors at either the gate level or the Register Transfer Level (RTL). At the gate level we injected bugs that complied with Abadir's error model [1], while those injected at the RTL were more complex functional errors (DLX contained real bugs). We collected input patterns for the benchmarks from several traces generated by verification (some of the traces were reduced by SimButramin), and a golden model was used to generate the correct output responses and state values for error diagnosis and correction. Note that the golden model can be a high-level behavior model because we do not need the simulation values for the internal signals of the circuit. The goal of the this experiment was to fix the layout of each benchmark so that the circuit produces correct output responses for the given input patterns. This is similar to the situation described in Section 2 where fixing the observed errors allows the silicon die to be used in further verification. If the repaired die fails further verification, new counterexamples will be used to refine the fix as described in the PAFER framework. The results are summarized in Table 2, where "#Patterns" is the number of input patterns used in each benchmark, and "#Resyn. cells" is the number of cells used by the resynthesis netlist. In order to measure the effects of our fix on important circuit parameters, we also report the changes in via count("#Vias"), wirelength ("WL"), and maximum delay ("Delay") after the layout is repaired. These numbers were collected after running NanoRoute in its ECO mode, and then they were compared to those obtained from the original layout. The maximum delay was reported by NanoRoute's timing analyzer.

The results in Table 2 show that our techniques can successfully repair logic errors for more than 70% of the benchmarks. We analyzed the benchmarks that could not be repaired and found that in those benchmarks, cells that produce the required signals were too far away and were excluded from our search. As a result, our resynthesis technique could not find valid fixes. In practice, it also means that the silicon die cannot be repaired via metal fix. The results also show that our error-repair techniques may change physical parameters such as via count, wirelength, and maximum delay. For example, the wirelength of SASC(GL1) increased by more than 1% after the layout was repaired. However, it is also possible that the fix we performed will actually improve these parameters. For example, the via count, wirelength, and maximum delay were all improved in DLX(GL2). In general, the changes in these physical parameters are typically small, showing that our error repair techniques have few side effects.

Benchmark	Bug description	#Patterns	#Resyn. cells	Changes after repair			Runtime (sec)
				#Vias	WL	Delay	
SASC(GL1)	Missing wire	90	2	0.29%	1.27%	-0.13%	9.9
SASC(GL2)	Incorrect gate	66	1	0.13%	0.33%	0.00%	4.4
EXU_ECL(GL1)	Incorrect gate	90	No valid fix was found				158.71
EXU_ECL(GL2)	Wrong wire	74	0	0.01%	0.03%	0.00%	145.3
Pre_norm(GL1)	Incorrect wire	46	2	0.10%	0.24%	-0.05%	38.92
DLX(GL1)	Incorrect gate	46	0	0.38%	0.02%	0.00%	17245
DLX(GL2)	Additional wire	33	0	-0.13%	-0.04%	-0.15%	12778
Pre_norm(RTL1)	Reduced OR replaced by reduced AND	672	3	0.19%	0.38%	0.57%	76.24
MD5(RTL1)	Incorrect state transition	201	3	0.02%	0.03%	-0.02%	29794
DLX(RTL1)	SLTIU inst. selects the wrong ALU operation	2208	No valid fix was found				12546
DLX(RTL2)	JAL inst. leads to incorrect bypass from MEM stage	1536	0	0.00%	0.00%	0.03%	8495
DLX(RTL3)	Incorrect forwarding for ALU+IMM inst.	1794	0	0.00%	0.00%	0.03%	13807
DLX(RTL4)	Does not write to reg31	1600	No valid fix was found				7723
DLX(RTL5)	If RT = 7 memory write is incorrect	992	0	0.00%	0.00%	0.00%	5771

Table 2: Functional error repair results. The bugs in the upper half were injected at the gate level, while those in the lower half were injected at the RTL. Some errors can be repaired by simply reconnecting wires and do not require the use of any spare cell, as shown in Column 4.

Benchmark	SymWire					SafeResynth				
	#Repaired	Metal segments affected			Runtime (sec)	#Repaired	Metal segments affected			Runtime (sec)
		Min	Max	Mean			Min	Max	Mean	
Stepper	**81**	6	33	**15.7**	0.03	**79**	14	53	**28.3**	4.68
SASC	**50**	8	49	**19.8**	0.79	**41**	2	48	**27.8**	3.32
EXU_ECL	**68**	7	42	**15.0**	1.13	**71**	14	831	**119.1**	23.02
MiniRISC	**58**	4	29	**13.7**	1.65	**57**	14	50	**28.1**	166
AC97_ctrl	**52**	9	26	**13.9**	3.26	**56**	14	53	**31.9**	68.02
USB_funct	**70**	7	36	**16.4**	1.84	**58**	16	74	**32.4**	157.52
MD5	**82**	7	30	**15.0**	1.83	**79**	13	102	**37.9**	2630
DLX	**64**	6	49	**15.8**	11.00	**67**	13	97	**40.2**	8257
PCI_bridge32	**42**	8	42	**16.6**	6.04	**32**	15	54	**31.2**	211.28
AES_core	**83**	5	32	**15.0**	2.53	**83**	12	64	**31.4**	285.58
WB_conmax	**84**	7	35	**16.0**	2.96	**46**	19	71	**35.2**	317.50
Alpha	**67**	9	41	**16.3**	12.32	**55**	11	101	**36.9**	85104
Ethernet	**36**	7	22	**13.4**	45.01	**18**	18	104	**46.6**	3714
DES_perf	**91**	7	1020	**36.7**	4.86	**76**	10	60	**29.0**	585.34

Table 3: Results of electrical error repair. 100 wires were randomly selected to be erroneous, and "#Repaired" is the number of errors that could be repaired by each technique. The number of metal segments affected by each technique is also shown.

6.2 Electrical Error Repair

We currently do not have access to tools that can identify electrical errors in a layout. Therefore, in this experiment we measure the effectiveness of our electrical error repair techniques by reporting the percentages of wires where at least one valid transformation can be found. To this end, we selected 100 random wires from each benchmark and assumed that the wires contained electrical errors. Next, we applied SymWire and SafeResynth to find layout transformations that can modify the wires to repair the errors. The results are summarized in Table 3. In the table, "#Repaired" is the number of wires that could be modified, and "Runtime" is the total runtime of analyzing all 100 wires. We also report the minimum, maximum and average numbers of metal segments affected by our error-repair techniques. These numbers include the segments removed and inserted due to the layout changes.

From the results, we observe that both SymWire and SafeResynth were able to modify more than half of the wires for most benchmarks, suggesting that they can effectively find layout transformations that change the electrical characteristics of the erroneous wires. In addition, the number of affected metal segments is often small, which indicates that both techniques have little physical impact to the chip, and the layout modifications can be implemented easily by FIB. The runtime comparison between these techniques shows that SymWire runs significantly faster than SafeResynth because symmetry detection for small sub-circuits is significantly faster than equivalence checking. However, SafeResynth is able to find and implement more aggressive layout changes for more difficult errors: as the results suggest, SafeResynth typically affects more metal segments than SymWire, producing more aggressive physical modifications. We also observe that SymWire seems to perform especially well for arithmetic cores such as MD5, AES_core, and DES_perf, possibly due to the large numbers of logic operations used in these cores. Since many basic logic operations are symmetric (such as AND, OR, XOR), SymWire is able to find many repair opportunities. On the other hand, SymWire seems to perform poorly for benchmarks which have high percentages of flip-flops, such as SASC, PCI_bridge32, and Ethernet. The reason is that SymWire is not able to find symmetries in flip-flops. As a result, if many wires only fanout to flip-flops, it will not be able to find fixes for those wires.

7. CONCLUSIONS

Due to the explosive increase in design complexity, more and more errors begin to escape pre-silicon verification and are discovered post-silicon. While most steps in the IC design flow have been highly automated, little effort has been devoted to the post-silicon debugging process, making it mostly ad-hoc and difficult. To address this problem, we propose the FogClear methodology that systematically automates the post-silicon debugging process, and it is powered by our new techniques and algorithms that enhances key steps in post-silicon debugging. An insight in these techniques is their comprehensive nature that allows the generation of various netlist or layout transformations, which is complemented by the intelligent use of pruning criteria derived from the restrictive physical constraints unique in post-silicon debugging. This innovation provides the foundation of our PAFER framework and the PARSyn algorithm that correct functional errors, as well as the SymWire and SafeResynth methods that repair electrical errors. With our techniques, post-silicon debugging is transformed from art into science. Our empirical results show that these techniques can repair a substantial number of errors in most benchmarks, demonstrating their effectiveness in facilitating the post-silicon debugging process. In addition to post-silicon debugging, FogClear can also be used to reduce the costs of respins: the fixes generated by FogClear only affect metal layers, allowing the reuse of transistor masks, thus reducing mask costs.

Acknowledgments. This work was partially funded by the NSF under Award 0448189.

8. REFERENCES

[1] M. S. Abadir, J. Ferguson and T. E. Kirkland, "Logic Verification via Test Generation", *IEEE TCAD*, pp. 138-148, Jan. 1988.

[2] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi and D. Miller, "A Reconfigurable Design-for-Debug Infrastructure for SoCs", *DAC'06*, pp. 7-12.

[3] K. Baker and J. V. Beers, "Shmoo Plotting: The Black Art of IC Testing", *IEEE Design and Test of Computers*, Vol. 14, No. 3, pp. 90-97, 1997.

[4] A. Balasinski, "Optimization of Sub-100-nm Designs for Mask Cost Reduction", *Journal of Microlithography, Microfabrication, and Microsystems*, Vol. 3, NO. 2, pp. 322-331, Apr. 2004.

[5] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing*, Kluwer, Boston, 2000.

[6] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?" *DAC'00*, pp. 693-698.

[7] K.-H. Chang, V. Bertacco and I. L. Markov, "Simulation-based Bug Trace Minimization with BMC-based Refinement", *ICCAD'05*, pp. 1045-1051.

[8] K.-H. Chang, I. L. Markov and V. Bertacco, "Post-Placement Rewiring and Rebuffering by Exhaustive Search For Functional Symmetries", *ICCAD'05*, pp. 56-63.

[9] K.-H. Chang, I. L. Markov and V. Bertacco, "Keeping Physical Synthesis Safe and Sound", *IWLS'06*, pp. 86-93.

[10] K.-H. Chang, I. L. Markov and V. Bertacco, "Safe Delay Optimization for Physical Synthesis", *ASPDAC'07*, pp. 628-633.

[11] K.-H. Chang, I. L. Markov and V. Bertacco, "Fixing Design Errors with Counterexamples and Resynthesis", *ASPDAC'07*, pp. 944-949.

[12] K.-H. Chang, V. Bertacco and I. L. Markov, "Simulation-based Bug Trace Minimization with BMC-based Refinement", *IEEE TCAD*, Vol. 26, No. 1, pp. 152-165, Jan. 2007.

[13] J. Ferguson, "Turning Up the Yield", *IEE Electronics Systems and Software*, pp. 12-15, June/July 2003.

[14] R. Goering, "Post-Silicon Debugging Worth a Second Look", *EETimes*, Feb. 05, 2007.

[15] W. Jiang, T. Marwah and D. Bouldin, "Enhancing Reliability and Flexibility of a System-on-Chip Using Reconfigurable Logic", *MWSCAS'05*, pp. 879-882.

[16] D. Josephson, "The Manic Depression of Microprocessor Debug", *ITC'02*, pp. 657-663.

[17] D. Josephson, "The Good, the Bad, and the Ugly of Silicon Debug", *DAC'06*, pp. 3-6.

[18] C.-H. Lin, Y.-C. Huang, S.-C. Chang, and W.-B. Jone, "Design and Design Automation of Rectification Logic for Engineering Change", *ASPDAC'05*, pp. 1006-1009.

[19] S.-J. Pan, K.-T. Cheng, J. Moondanos, and Z. Hanna, "Generation of Shorter Sequences for High Resolution Error Diagnosis Using Sequential SAT", *ASPDAC'06*, pp. 25-29.

[20] S. Safarpour, A. Veneris, and H. Mangassarian, "Trace Compaction using SAT-based Reachability Analysis", *ASPDAC'07*, pp. 932-937.

[21] A. Veneris and I. N. Hajj, "Design Error Diagnosis and Correction via Test Vector Simulation", *IEEE TCAD*, pp. 1803-1816, Dec. 1999.

[22] H. Xiang, L.-D. Huang, K.-Y. Chao, and M. D. F. Wong, "An ECO Algorithm for Resolving OPC and Coupling Capacitance Violations", *ASICON'05*, pp. 784-787.

[23] Y.-S. Yang, S. Sinha, A. Veneris and R. E. Brayton, "Automating Logic Rectification by Approximate SPFDs", *ASPDAC'07*, pp. 402-407.

[24] J. Zhang, S. Sinha, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Simulation and Satisfiability in Logic Synthesis", *Proc. IWLS'05*, pp. 161-168

[25] International Technology Roadmap for Semiconductors 2005 Edition, `http://www.itrs.net`

[26] `http://www.dafca.com/`

[27] `http://www.opencores.com/`

[28] `http://opensparc-t1.sunsource.net/`