InVerS: An Incremental Verification System with Circuit Similarity Metrics and Error Visualization

Kai-hui Chang, David A. Papa, Igor L. Markov and Valeria Bertacco
Department of EECS, University of Michigan at Ann Arbor
E-mail: {changkh, iamyou, imarkov, valeria}@eecs.umich.edu

Abstract

Dramatic increases in design complexity and advances in IC manufacturing technology affect all aspects of circuit performance and functional correctness. As interconnect increasingly dominates delay and power at the latest technology nodes, much effort is invested in physical synthesis optimizations, posing great challenges in validating the correctness of such optimizations. Common design methodology delays the verification of physical synthesis transformations until the completion of the design phase. However, this approach is not sustainable because the isolation of potential errors becomes extremely challenging in current complex design efforts. In addition, the lack of interoperability between verification and debugging tools greatly limits engineers' productivity. Since the design's functional correctness should not be compromised, considerable resources are dedicated to checking an ensuring correctness at the expense of improving other aspects of design quality. To address these challenges, we propose a fast incremental verification system for physical synthesis optimizations, InVerS, which includes capabilities for error detection, diagnosis, and visualization. This system helps engineers to discover errors earlier, simplifies error isolation and correction, thus reducing verification effort and enabling more aggressive optimizations to improve performance.

1. Introduction

The growth in complexity of digital designs poses greater challenges in verifying the functional correctness of a circuit. As a result, digital systems are commonly released with latent bugs, and the number of such bugs is growing larger for each new design, as can be observed from publicly available errata documents by any major semiconductor vendor. The verification problem is further exacerbated by the growing dominance of interconnect in delay and power of modern designs, which requires tremendous physical synthesis effort and even more powerful optimizations such as retiming [9]. Traditional techniques address this problem by checking the equivalence between the original design and the optimized version that has undergone physical optimizations. This approach, however, only verifies the equivalence of two versions of the design after a number, or possibly all, of the transformations and optimizations have been completed. Unfortunately, such an approach is not sustainable in the long term because it makes the identification, isolation, and correction of errors introduced by such transformations extremely difficult and time-consuming. On the other hand, performing traditional equivalence checking after each circuit transformation is too demanding. Since functional correctness is the most important aspect of high-quality designs, large efforts are currently devoted to verification and debugging, expending resources that could have otherwise been dedicated to improve other aspects of performance. To this end, verification has become the bottleneck that limits achievable optimizations and the features that can be included in a design [4], slowing down the evolution of the overall quality of electronic designs.

Given the current practice, it is crucial to address the verification bottleneck to improve design quality. We advocate not only investing in the performance of verification algorithms and tools, but also revising the design methodology to ease the burden on verification effort. To this end, we propose an Incremental Verification System (InVerS) that signals design errors earlier than existing methodologies. The high performance of our equivalence verification solution allows quick evaluation of the correctness of each design transformation. When an error is detected, InVerS provides a counterexample so that the designer can analyze the error directly. Our technique also suggests the most probable location and source of the error, pinpointing, in most cases, the specific transformation responsible for it. This information greatly facilitates error diagnosis. To further improve designers' productivity, InVerS provides an intuitive Graphical User Interface (GUI). Our implementation is built using the OpenAccess database [13] and uses the OpenAccess Gear (OAGear) programmer's toolkit, so that we can seamlessly integrate design, verification and debugging activities into the same framework. This framework is highly flexible and can easily be enhanced in the future.

The contributions of this work include: (1) InVerS, an incremental equivalence verification methodology that enhances the accuracy of error detection; (2) an innovative and scalable metric, called the *similarity factor*, that quickly pinpoints potential spots in a design that could be hiding bugs; (3) a fast simulator based on OpenAccess that is 100 times faster on large designs than previous implementations; and (4) a GUI for InVerS with data visualization that improves the usability of our tools. Our techniques can greatly improve design quality because: (1) the resources and effort saved in verifying the correctness of physical optimizations can be redirected to improve other aspects of the design, such as reliability and performance; and (2) more aggressive changes to the circuit can be applied, such as retiming optimizations and design-for-verification (DFV) techniques.

The rest of this paper is organized as follows. In Section 2 we review previous work and background material. We describe our incremental verification system in detail in Section 3. In Section 4 we present our productivity enhancing GUI and error visualization tools. Experimental results are shown in Section 5, and Section 6 concludes this paper.

2. Background

InVerS addresses the verification problem created by logic changes to the netlist. To understand the problem better, we describe two commonly used optimization techniques that make changes to the

Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED'07)
0-7695-2795-7/07 $20.00 © 2007 IEEE

netlist, physical synthesis and retiming. We then briefly explain the challenges imposed by these optimization techniques to verification. Next, we introduce OpenAccess and OAGear, the infrastructure used to develop our tool. Finally, the error model used in our experiments is described.

2.1. Physical Synthesis Flows

Post-placement optimizations have been studied and used extensively to improve circuit parameters, and such techniques are often called physical synthesis. In addition, it is sometimes necessary to change the layout manually in order to fix bugs or optimize specific objectives; this process is called Engineering Change Order (ECO). Physical synthesis is commonly performed using the following flow: (1) perform accurate analysis of the optimization objective, (2) select gates to form a region for optimization, (3) resynthesize the region to optimize the objective, and (4) perform legalization to repair the layout. The work by Lu *et al.* [8] and Changfan *et al.* [3] are all based on this flow.

Since bugs may be introduced by the circuit modifications, verification must be performed to ensure the correctness of the circuit. However, verification is typically slow; therefore, it is often performed after hundreds of optimizations. As a result, it is difficult to identify the circuit modification that introduced the bug. In addition, debugging the circuit at this design stage is often difficult because engineers are unfamiliar with the automatically generated netlist. As we will show later, InVerS addresses these problems by providing a fast incremental verification technique and an integrated error visualization tool.

2.2. Retiming

Retiming is a sequential logic optimization technique that repositions the registers in a circuit while leaving the combinational cells unchanged [9]. It is often used to minimize the number of registers in a design or to reduce a circuit's delay. For example, the circuit in Figure 1(b) is a retimed version of the circuit in Figure 1(a) that optimizes delay. Although retiming is a powerful technique, ensuring its correctness imposes a serious problem on verification because sequential equivalence checking is orders of magnitude more difficult than combinational equivalence checking [6]. As a result, the runtime of sequential verification is often much longer than that of combinational verification, if it ever finishes. As we will show in Section 3.4, our technique can be extended to sequential verification for retiming.

2.3. OpenAccess and OAGear

OpenAccess is a VLSI design data model with a standardized database representation and a programming API. This common design model with persistent storage provides a convenient platform on which to build incremental algorithms, so we chose to leverage the power of the OpenAccess database in our work. OAGear has an existing GUI based on the OpenAccess data model, and it provides additional infrastructure that we utilize in building our new user interface and visualization tools.

2.4. Error Model

When evaluating our tool in Section 5, we need to inject errors into existing circuits. In order to inject more realistic errors, we adopted a frequently used error model based on Abadir's work [1]. The model consists of the following errors: (1) "wrong gate" replaces one gate by another one with the same number of inputs; (2) "extra/missing wire" use more or fewer inputs for a gate; (3) "wrong input" connects an input of a gate to a wrong driver; and (4) "extra/missing gate" incorrectly inserted or removed a gate. To inject an error, we first randomly select one gate in the circuit. Next, we choose an error type randomly and then change the gate or its connections accordingly.

3. Incremental Verification

We provide a robust incremental verification package that is composed of a logic simulator, a SAT-based formal equivalence-checker, routines to compute our new similarity metric between a circuit and its revision, and new visualization tools to aid users of our proposed incremental verification methodology. Our equivalence checker first uses random simulation to quickly detect signals that are not equivalent. For the signals that cannot be distinguished by random simulation, SAT-based equivalence checking is used, and counterexamples found during SAT-solving are reused as additional simulation patterns to distinguish more signals [10]. This implementation and its interface is used to support incremental verification, as explained below.

3.1. Fast Simulation Algorithms

Our simulator first extracts logic information, an And-Inverter-Graph (AIG), for each cell used in the design from OAGear's Func(tional) package. Next, we simulate all possible input combinations of each cell to construct its truth-table. Using such look-up tables during simulation is far more efficient than traversing AIGs of individual cells. To further improve speed, our simulator employs bit-parallel simulation (32 or 64 patterns simulated at once depending on the definition of `SimulationVector`) and treats most common gate types as special cases. In order to efficiently simulate patterns with different event activity, we implemented an oblivious algorithm as well as an event-driven algorithm [7]. We observe that our simulation algorithm runs 100 times faster than the existing simulator in OAGear on large designs.

3.2. SAT-based Equivalence Checking

Our equivalence checker first generates the CNF of every cell in the library. This is accomplished by traversing the AIG of each cell and converting the ANDs and INVERTERs to their corresponding circuit-CNFs. Next, we build a miter for the signals to be checked for equivalence and convert it to CNF. A miter is a circuit consisting of an XOR gate combining the signals and their fanin cones with depth such that the inputs to each cone are the same. We set the output of the miter to 1 and use MiniSAT [5] to determine satisfiability. If the CNF is not satisfiable, the signals are equivalent, alternatively, a counterexample is returned by the SAT solver. We employ a simple interface to a SAT-solver so that MiniSAT can be easily replaced and CNF conversion can be improved. The user can adjust the number of initial patterns used by random simulation. Setting that number to 0 turns off random simulation and resorts to SAT-based equivalence checking.

3.3. New Metric: Similarity Factor

We define an estimate of the similarity between two netlists, ckt_1 and ckt_2, that utilizes fast simulation, called the *similarity factor*. This metric is based on simulation signatures of individual signals, i.e. the k-bit sequences holding signal values computed by simulation on each of k input patterns (e.g., k=1024). Let N be the total number of signals (wires) in both circuits. Out of those N signals, we distinguish M *matching* signals — a signal is considered matching if and only if both circuits include signals with an

identical signature. The similarity factor between ckt_1 and ckt_2 is then M/N. In other words:

$$Similarity\ factor = \frac{number\ of\ matching\ signals}{total\ number\ of\ signals} \quad (1)$$

We also define the *difference factor* as $(1 - similarity\ factor)$.

EXAMPLE 1. *Suppose that netlist ckt_1 has 3 signals, whose signatures are 1, 2, and 3; netlist ckt_2 has 3 signals, whose signatures are 1, 2, and 4. Since the total number of signatures is 6 and the number of matching signals is 4, the similarity factor is 4/6 = 0.67 and the difference factor is 1 - 4/6 = 0.33.*

Intuitively, the similarity factor of two identical circuits should be 1. If a circuit is changed slightly but is still mostly equivalent to the original version, then its similarity factor should drop only slightly. However, if the change greatly affects the circuit's function, the similarity factor can drop significantly, depending on the number of signals affected by the change. The new similarity metric relies on simulation but not on SAT solvers, allowing fast computation. However, two equivalent circuits may be dissimilar, e.g., a Carry-Look-Ahead adder and a Kogge-Stone adder. Therefore, the similarity factor should be used in incremental verification and cannot replace traditional verification techniques.

The similarity factor can also be used to support error diagnosis for a given bug trace. Since the effect of a bug is to change the signatures of all downstream signals, the signatures will not match those in the original circuit. Using hash-based signature-matching, we can quickly identify such downstream logic and potentially map it to Verilog code. Moreover, it is possible to identify gates whose inputs have matching signatures while outputs do not. The user interface of our system highlights such gates as potential error locations, which is described in Section 4.

3.4. Sequential Verification for Retiming

A signature represents a fraction of a signal's truth table, which in turn describes the information flow within a circuit. While retiming may change the clock cycle that certain signatures are generated, because combinational cells are preserved, it should not change which signatures are generated. Figure 1 shows a retiming example adopted from Shenoy's work [9], where (a) is the original circuit and (b) is the retimed circuit. A comparison of signatures between the circuits shows that the signatures in (a) also appear in (b), although the cycles in which they appear may be different. For example, the signatures of wire w (bold-faced) in the retimed circuit appear one cycle earlier than those in the original circuit because the registers were moved later in the circuit. Otherwise, the signatures of (a) and (b) are identical.

Based on this observation, we extend our similarity factor to sequential verification as follows. Given circuit ckt_1 and its retimed revision ckt_2, we perform sequential simulation for i cycles using k random input patterns at each cycle. Let n be the total number of signals used by the two circuits, then the total number of signatures of the two circuits, denoted as N, will be $n \times i$. Out of those N signatures, we distinguish M matching signatures, which are signatures that exist in both ckt_1 and ckt_2, and define the similarity factor between ckt_1 and ckt_2 as M/N. Similar to verification of combinational circuits, a sudden drop in the similarity factor after retiming would indicate potential bugs.

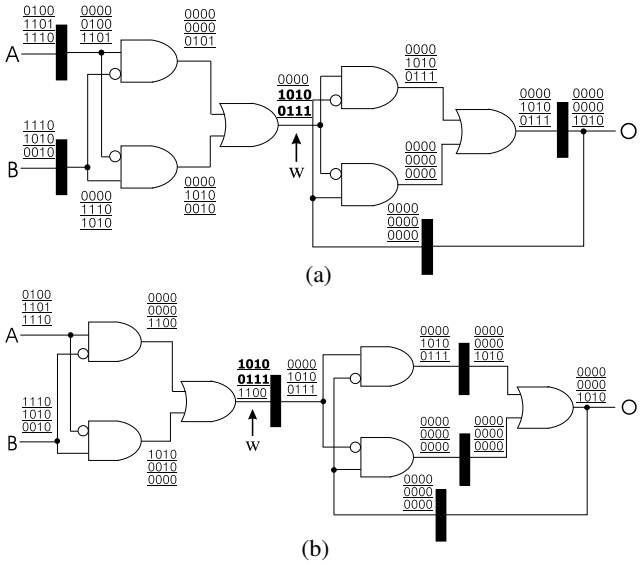


Figure 1. A retiming example: (a) is the original circuit, and (b) is its retimed version. The tables above the wires show their signatures, where the n^{th} row is for the n^{th} cycle. Four traces are used to generate the signatures, producing four bits per signature. Registers are represented by black rectangles, and their initial states are 0. As wire w shows, retiming may change the cycle that signatures appear, but it does not change the signatures (signatures shown in boldface are identical).

3.5. Overall Verification Methodology

As mentioned in Section 1, traditional verification is typically performed after a batch of circuit modifications because it is very demanding and time consuming. As a result, once a bug is found, it is often difficult to isolate the change that introduces the bug because hundreds or thousands of changes have been made. Similarity factor addresses this problem by pointing out the changes that might have corrupted the circuit. As described in previous subsections, a change that greatly affects the circuit's function will probably cause a sudden drop in the similarity factor. By monitoring the change in similarity factor after every circuit modification, engineers will be able to know when a bug might have been introduced and traditional verification should be performed.

In InVerS, we apply incremental verification as follows:

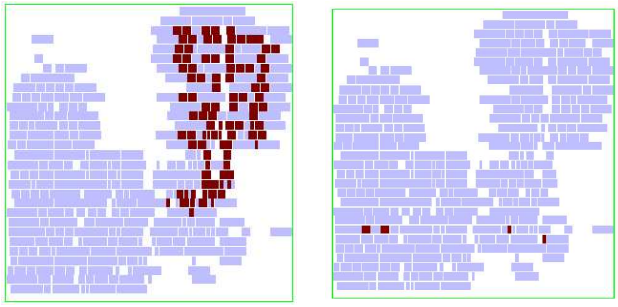
1. After each change to the circuit, the similarity factor between the new and the original circuit is calculated. Running average and standard deviation of the past 30 similarity factors are used to determine whether the current similarity factor has dropped significantly. Empirically, we found that if the current similarity factor drops below the average by more than two standard deviations, then it is likely that the change introduced a bug.
2. When similarity factor indicates a potential problem, traditional verification should be performed to verify the correctness of the executed circuit modification.
3. If verification fails, our error visualization tools can be used to debug the errors by highlighting the gates producing erroneous signals.

As Section 5 shows, the similarity factor has high accuracy for practical designs and allows our verification methodology to achieve significant speed-up over traditional verification techniques.

4. Efficient User Interface

The usability of EDA tools affects the productivity of VLSI verification engineers. To improve productivity, we developed a novel user interface, as well as several visualization tools, that improve the usability of our verification methodology.

InVerS adds several new use-cases to the OAGear GUI – Bazaar [13]. One simple yet powerful use-case is to run simulation and display the results using an efficient simulation algorithm developed on our own. Simulation signatures are subsequently used to filter potentially equivalent signals. Any signals that cannot be differentiated by simulation signatures alone will be checked for equivalence using SAT-based exhaustive techniques. Performing equivalence checking and displaying counter examples upon failure is another use-case. Finally, if transforming a circuit produces a logically different circuit, the gates that create this difference can be highlighted on a layout or schematic to graphically display the equivalence data.



(a) (b)

Figure 2. Our similarity layout viewer for design SASC with bug-related data shown in red (darker color): (a) one bug injected; highlighted gates drive unmatched signals; (b) 5 unrelated bugs injected; highlighted gates drive unmatched signals, but all of their inputs are matched; 4 bugs are identified, and one is masked.

4.1. Error Visualization

The computation of similarity factor works by matching signals from two revisions of a design. Naturally, this process also identifies those nets for which there are no matches in the previous design version. Those nets are of interest to the designer because they are responsible for the change in circuit behavior. One way to visually represent the data of nets that match versus those do not match is to draw the layout of gates driving those nets using different colors. Similarly, the schematic can be displayed using different colors for gates driving non-matching nets than for those driving matching ones. Both of these techniques show the designer a large amount of data in a familiar form with an instant recognition of which gates are different, as the similarity layout view in Figure 2(a) shows.

Another way to quickly ascertain what insights the similarity data contains is to highlight those gates whose inputs match, but whose outputs do not match. These gates very likely correspond to a bug, as is shown in Figure 2(b). However, fixing these bugs may unmask other bugs, which is illustrated by the fact that we found only 4 of the 5 injected bugs in this example. Looking at these two types of bug visualizations will allow the designer to more efficiently reason about circuits with errors, and ideally, to quickly locate and resolve errors and what caused them.

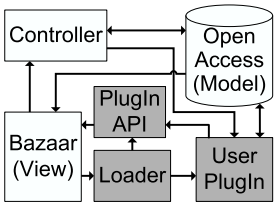


Figure 3. Design of plug-in interface incorporated into Bazaar.

4.2. Software Design

To minimize the impact of integrating our tools into Bazaar and improve their usability, we created a plug-in for InVerS that can be loaded optionally by the user at runtime. Figure 3 shows the overall design of Bazaar with its new plug-in interface shaded.

Bazaar adheres to the Model-View-Controller design pattern [2], where an integrated circuit is represented in an underlying data model (OpenAccess) and can be viewed and modified using a GUI (Bazaar). To facilitate dynamic loading of plug-ins, we added two additional components to the design above, a plug-in API object (piAPI) and a loader object.

When the user loads a plug-in into Bazaar, the loader creates a piAPI object and supplies it with a pointer to Bazaar. A user-provided function `MyPlugIn::load(auto_ptr<piAPI>)` is then called by the loader. This function takes possession of the piAPI object and creates menus, toolbars, windows, and commands in Bazaar. Thus, we decouple Bazaar from plug-ins since Bazaar cannot access the piAPI object, and the plug-in can only access Bazaar through the provided API. This allows Bazaar to safely load various configurations of plug-ins on demand.

4.3. Software Engineering Details

Our simulator and equivalence checker are both implemented natively in OpenAccess and follow the OAGear coding standards. Our new software includes documentation and is supplied with regression tests. Additionally, we provide a variety of convenient ways to use these tools including standalone binaries, point-and-click use-cases in Bazaar, and a library API. To encourage adoption of our new tools, their interfaces are designed to be compatible with the existing versions. Our package has some limitations, specifically: (1) it only supports the datamodel of OA's block domain as opposed to the module or occurrence domains, and (2) the netlist must be mapped to a cell library.

5. Experimental Results

We implemented InVerS using OAGear 0.96 and OpenAccess 2.2. Our testcases are selected from IWLS'05 benchmarks [12] based on designs from OpenCores suites, whose characteristics are summarized in Table 1. In the table, the average level of logic is calculated by averaging the logic level of 30 randomly selected gates. The number of levels of logic can be used as an indication of the circuit's complexity. All our experiments were conducted on an AMD Opteron 880 Linux workstation. The resynthesis package used in our experiments is ABC from UC Berkeley [11], which is based on AIGs. In this section, we first evaluate the accuracy of the similarity factor, and then we investigate the impact of cell count and level of logic on the similarity factor. The second experiment evaluates the effectiveness and efficiency of our incremental verification methodology.

Evaluation of the similarity factor: in our first experiment, we perform two types of circuit modifications to evaluate the effec-

Benchmark	Similarity factor (%)									
	One Error injected				Resynthesized				d_1	d_2
	$Mean_e$	Min_e	Max_e	SD_e	$Mean_r$	Min_r	Max_r	SD_r		
USB_PHY	98.897	91.897	99.822	1.734	99.849	99.019	100.000	0.231	0.969	4.128
SASC	97.995	90.291	99.912	2.941	99.765	99.119	100.000	0.234	1.115	7.567
I2C	99.695	98.583	100.000	0.339	99.840	99.486	100.000	0.172	0.567	0.843
SPI	99.692	96.430	99.985	0.726	99.906	99.604	100.000	0.097	0.518	2.191
TV80	99.432	94.978	100.000	1.077	99.956	99.791	100.000	0.050	0.930	10.425
MEM_CTRL	99.850	97.699	100.000	0.438	99.984	99.857	100.000	0.027	0.575	4.897
PCI_BRIDGE32	99.903	97.649	99.997	0.426	99.978	99.941	100.000	0.019	0.338	3.878
AES_CORE	99.657	98.086	99.988	0.470	99.990	99.950	100.000	0.015	1.372	21.797
WB_CONMAX	99.920	99.216	99.998	0.180	99.984	99.960	100.000	0.012	0.671	5.184
DES_PERF	99.942	99.734	100.000	0.072	99.997	99.993	100.000	0.002	1.481	23.969

Table 2. Statistics of similarity factors for different types of circuit modifications. Thirty tests were performed in this experiment, whose means, minimal values (Min), maximum values (Max), and standard deviations (SD) are shown. The last two columns show the standardized differences in the means: d_1 is calculated using the average of both SD_e and SD_r, while d_2 uses only SD_r.

Benchmark	Cell count	Ave. Lev. of logic	Function
USB_PHY	546	4.7	USB 1.1 PHY
SASC	549	3.7	Simple Asynchronous Serial Controller
I2C	1142	5.5	I2C Master Controller
SPI	3227	15.9	SPI IP
TV80	7161	18.7	8-Bit Microprocessor
MEM_CTRL	11440	10.1	WISHBONE Memory Controller
PCI_BRIDGE32	16816	9.4	PCI bridge
AES_CORE	20795	11.0	AES Cipher
WB_CONMAX	29034	8.9	WISHBONE Conmax IP Core
DES_PERF	98341	13.9	DES Cipher

Table 1. Benchmark characteristics.

tiveness of the similarity factor. In the first type, we randomly inject an error into the circuit using the technique described in Section 2.4. This mimics the situation where a bug has been introduced. In the second type, we extract a subcircuit from the benchmark, which is composed of 2-20 gates, and perform resynthesis of the subcircuit using ABC with the "resyn" command [11]. This is similar to the physical synthesis or ECO flow described in Section 2.1, where gates in a small region of the circuit are changed. We then calculate the similarity factor after each circuit modification for both types of circuit modifications and compare their difference. Thirty tests were performed in this experiment, and the results are summarized in Table 2. From the results, we observe that both types of circuit modifications lead to decreases in similarity factor. However, the decrease is much more significant when an error is injected. As d_1 shows, the standardized differences in the means of most benchmarks are larger than 0.5, indicating that the differences are statistically significant. Since resynthesis tests represent the norm and error-injection tests represent anomalies, we also calculate d_2 using only SD_r. As d_2 shows, the mean similarity factor drops more than two standard deviations when an error is injected for most benchmarks. This result shows that the similarity factor is effective in predicting whether a bug has been introduced by the circuit modification. Nonetheless, in all benchmarks, the maximum similarity factor for error-injection tests is larger than the minimum similarity factor for resynthesis tests, suggesting that the similarity factor cannot replace traditional verification and should be used as an auxiliary technique.

The impact of cell count on the similarity factor: in order to study other aspects that may affect the similarity factor, we further analyze our results by plotting the factors against the cell counts of the benchmarks. To make the figure clearer, we plot the difference factor instead of the similarity factor. We notice that by construction, the difference factor tends to reduce with the increase in design size, which makes the comparison among different benchmarks difficult. In order to compensate this effect, we assume that the bug density is 1 bug per 1,000 gates and adjust our numbers accordingly. The plot is shown in Figure 4, where the triangles represent data points from error-injection tests, and the squares represent resynthesis tests. The linear regression lines of two data sets are also shown. From the figure, we observe that the difference factor tends to increase with the cell count for error-injection tests. The increase for resynthesis tests, however, is insignificant. As a result, the difference factor of error-injected circuits (triangle data points) will grow faster than that of resynthesized circuits (square data points) when cell count increases, creating larger discrepancy between them. This result shows that the similarity factor will drop more significantly for larger designs, making it more accurate when applied practical designs, which often have orders of magnitude more objects than benchmarks used in our tests.

The impact of level of logic on the similarity factor: here we perform similar analysis using the number of levels of logic as the independent variable. The slopes of the linear regression lines for the error-injection tests and the resynthesis tests are 0.236 and 0.012, respectively. The difference in slopes shows that the difference factor grows faster when the number of levels of logic increases, indicating that the similarity factor will be more effective when designs become more complicated. This behavior is preferable because complicated designs are often more difficult to verify.

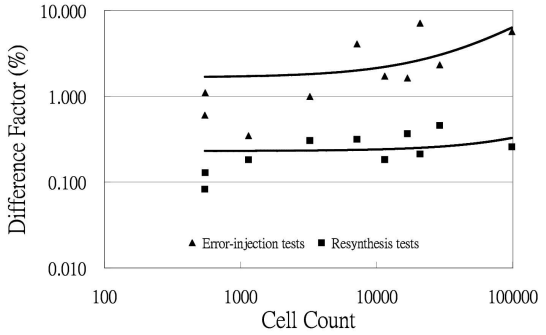


Figure 4. The relationship between cell count and the difference factor. The linear regression lines of the datapoints are also shown.

To study the impact of the number of levels of logic on the difference factor within a benchmark, we plotted the difference factor

against the number of levels of logic using benchmark des_perf in Figure 5. The logarithmic regression line for the error-injection tests are also shown. As the figure suggests, the difference factor decreases with the increase in the number of levels of logic. The reason is that gates with smaller numbers of levels of logic have larger downstream logic, therefore larger numbers of signatures will be affected. As a result, the difference factor will be larger. That the variance explained is large (0.7841) suggests that this relation is strong. However, some benchmarks do not exhibit this trend. For example, the variance explained for benchmark TV18 is only 0.1438. For benchmarks that exhibit this trend, the similarity factor provides a good predication of the location of the bug: a larger drop in the similarity factor indicates that the bug is closer to primary inputs.

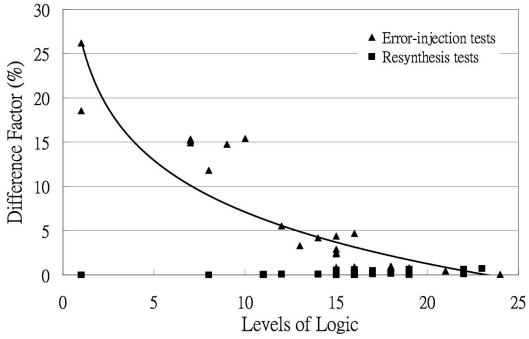


Figure 5. The relationship between the number of levels of logic and the difference factor in benchmark DES_PERF. The x-axis is the level of logic that the circuit is modified. The logarithmic regression line for the error-injection tests is also shown.

Evaluation of the incremental verification methodology: our second experiment evaluates the effectiveness of our incremental verification methodology described in Section 3.5. In this experiment, we assume that there is 1 bug per 100 circuit modifications, and then we calculate the accuracy of our methodology. We also report the runtime for calculating the similarity factor and the runtime for equivalence checking of each benchmark. Since most circuit modifications do not introduce bugs, we report the runtime when equivalence is maintained. The results are summarized in Table 3. From the results, we observe that our methodology has high accuracy for most benchmarks. In addition, the results show that calculating the similarity factor is significantly faster than performing equivalence checking. Take the largest benchmark (DES_PERF) for example, calculating the similarity factor takes less than 1 second, while performing equivalence checking takes about 78 minutes. Due to the high accuracy of the similarity factor, our incremental verification technique identifies more than 99% of errors, rendering equivalence checking unnecessary in those cases and providing a more than 100X speed-up.

6. Conclusions

In this work we developed a novel incremental equivalence verification system, InVerS, with a particular focus on improving design quality and engineer's productivity. The high performance of InVerS allows designers to invoke it frequently, after each circuit transformation, and thereby detect errors sooner, when these errors can be more easily pinpointed and resolved. The scalability of InVerS stems from the use of a novel fast simulator, which can efficiently calculate a "similarity factor" metric to spot potential differences between two versions of a design. The areas

Benchmark	Cell count	Accuracy	Runtime(s)	
			EC	SF
USB_PHY	546	92.70%	0.19	<0.01
SASC	549	89.47%	0.29	<0.01
I2C	1142	95.87%	0.54	<0.01
SPI	3227	96.20%	6.90	<0.01
TV80	7161	96.27%	276.87	0.01
MEM_CTRL	11440	99.20%	56.85	0.03
PCI_BRIDGE32	16816	99.17%	518.87	0.04
AES_CORE	20795	99.33%	163.88	0.04
WB_CONMAX	29034	92.57%	951.01	0.06
DES_PERF	98341	99.73%	4721.77	0.19

Table 3. The accuracy of our incremental verification methodology. 1 bug per 100 circuit modifications is assumed in this experiment. Runtime for similarity-factor calculation (SF) and equivalence checking (EC) are also shown.

where we detect a low similarity are spots potentially hiding functional bugs that can be subjected to more expensive formal techniques or visually inspected. Therefore, we provide a user interface to improve the usability of our methodology and support the designer in the debugging task. Part of this user interface is our error visualization tool that graphically reveals the difference between two circuits, allowing the designer to pinpoint the root cause of the bugs more easily. The experimental results show that InVerS achieves a hundred-fold runtime speed-up on large designs compared to traditional techniques for similar verification goals. Our methodology and algorithms promise to decrease the number of latent bugs released in future digital designs and to facilitate more aggressive performance optimizations, thus improving the quality of electronic design in several categories.

7. References

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

[2] S. Burbeck, "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller", `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`

[3] C. Changfan, Y. C. Hsu and F. S. Tsai, "Timing Optimization on Routed Designs with Incremental Placement and Routing Characterization", *IEEE Trans. on CAD*, Feb. 2000, pp. 188-196.

[4] I. Chayut, "Next-Generation Multimedia Designs: Verification Needs," DAC'06, Section 23.2, `http://www.dac.com/43rd/43talkindex.html`

[5] N. Eén and N. Sörensson, "An Extensible SAT-solver", *Theory and Applications of Satisfiability Testing, SAT*, 2003, pp. 502-518.

[6] J.-H. R. Jiang and R. K. Brayton, "On the Verification of Sequential Equivalence", *IEEE Transactions on Computer-Aided Design*, Jun. 2003, pp. 686-697.

[7] D. M. Lewis, "A Hierarchical Compiled-Code Event-Driven Logic Simulator", *IEEE Transactions on Computer-Aided Design*, Jul. 1987, pp.601-617.

[8] A. Lu, H. Eisenmann, G. Stenz and F. M. Johannes, "Combining Technology Mapping with Post-Placement Resynthesis for Performance Optimization", *ICCD'98*, pp. 616-621.

[9] N. Shenoy and R. Rudell, "Efficient Implementation of Retiming", *ICCAD'94*, pp. 226-233.

[10] J. Zhang, S. Sinha, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske, "Simulation and Satisfiability in Logic Synthesis", *IWLS* 2005, pp. 161-168.

[11] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 51205. `http://www-cad.eecs.berkeley.edu/~alanmi/abc/`

[12] `http://iwls.org/iwls2005/benchmarks.html`

[13] `http://www.si2.org/`