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
{changkh, iamyou, imarkov, valeria}@eecs.umich.edu

April 25, 2008

Abstract

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 methodologies postpone the verification of physical synthesis transformations until the completion of the design phase. The design flow is no longer sustainable because isolating potential errors produced by these physical transformations becomes extremely challenging at the late design stages. To address these issues, we propose a fast incremental verification system for physical synthesis optimizations, InVerS, which includes capabilities for error detection, diagnosis, and visualization. InVerS is based on a simple yet effective circuit similarity metric to quickly help engineers identify potential errors earlier in the development, and it resorts to traditional verification only when necessary to ensure the completeness of the verification flow. InVerS also provides an error visualization interface to simplify error isolation and correction, thereby reducing verification effort and enabling more aggressive optimizations to improve design performance.

Keywords: functional verification, debugging, equivalence checking

1 Introduction

The complexity growth of digital designs poses increasing challenges to the functional verification 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 [11] and even more powerful optimizations such as retiming [6]. Given that bugs still appear in many EDA tools today [9], it

is important to verify the correctness of the performed optimizations. Traditional techniques address this verification problem by checking the equivalence between the original design and the optimized version. 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, a large amount of effort is currently devoted to verification and debugging, expending resources that could have otherwise been dedicated to improve performance. Because of this, verification has become the bottleneck that limits which novel features that can be included in a design [3], slowing down the evolution of the overall quality of electronic designs.

Given the current practice, it is crucial to address this verification bottleneck to improve design quality. Recent advancements to this end often focus on improving the performance of the verification tool itself. For example, several techniques that use simulation to accelerate SAT or BDD-based equivalence checking have been proposed [7]. 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 and debugging effort. We propose an Incremental Verification System (InVerS) capable of exposing design errors earlier on during the optimization flow, hence facilitating debugging. 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. To further improve designers' productivity, InVerS provides an intuitive Graphical User Interface (GUI). Our implementation is built using the OpenAccess database [10] 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 verification methodology that enhances the accuracy of error detection; (2) an innovative and scalable metric, called the *similarity factor*, that quickly pinpoints potential bug locations; and (3) 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. Experimental results are shown in Section 4, and Section 5 concludes this paper.

2 Background

InVerS addresses the functional verification of incremental netlist transformations. To understand the problem better, we describe two common optimization techniques: physical synthesis and retiming. We then briefly explain the challenges to verification imposed by these techniques.

2.1 Physical Synthesis Flows

Post-placement optimizations have been studied and used extensively to improve circuit parameters such as power and timing, and these 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.

Given that subtle and unexpected bugs still appear in physical synthesis tools today [9], verification must be performed to ensure the correctness of the circuit. However, verification is typically a time-consuming process; therefore, it is often postponed until hundreds of optimizations have been completed. Consequently, if an error is detected, it is difficult to pinpoint the specific circuit modification that introduced the bug. In addition, debugging a circuit at this design stage is often difficult because engineers are unfamiliar with the automatically generated netlist. As we 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 relocates registers in a circuit while leaving the combinational cells unchanged [6]. It is often used to minimize the number of registers in a design or to reduce a circuit's delay. Although retiming is a powerful technique, ensuring its correctness is an even more complex verification problem because sequential equivalence checking is much more difficult than combinational equivalence checking [5]. As a result, if the analysis terminates, the runtime of sequential verification is often much larger than that of combinational verification. In this paper we propose new techniques that extend our previous work [2] to address the sequential verification problem for retiming.

3 Incremental Verification

We present an incremental verification package that is composed of a logic simulator, a SAT-based formal equivalence checker, our innovative similarity metric between a circuit and its revision, and new visualization tools to aid users of our proposed incremental verification methodology. In this section we define our similarity metric, illustrate the error visualization interface, and then describe our overall verification methodology.

3.1 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:

$$similarity\ factor = \frac{number\ of\ matching\ signals}{total\ number\ of\ signals} \tag{1}$$

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

Example 1 *Consider the two netlists shown in Figure 1, where the signatures are shown above the wires. There are 10 signals in the netlists, and 7 of them are matching. As a result, the similarity factor is 7/10= 70%, and the difference factor is 1 - 7/10 = 30%.*

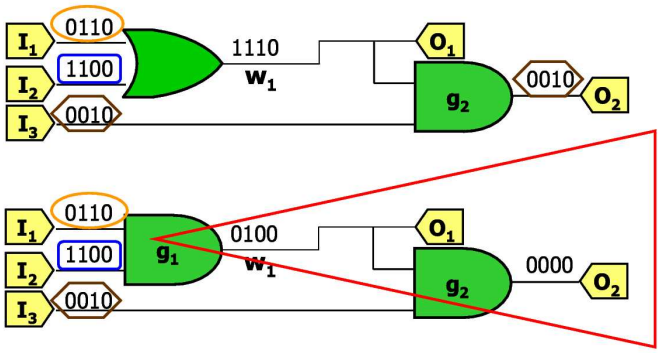
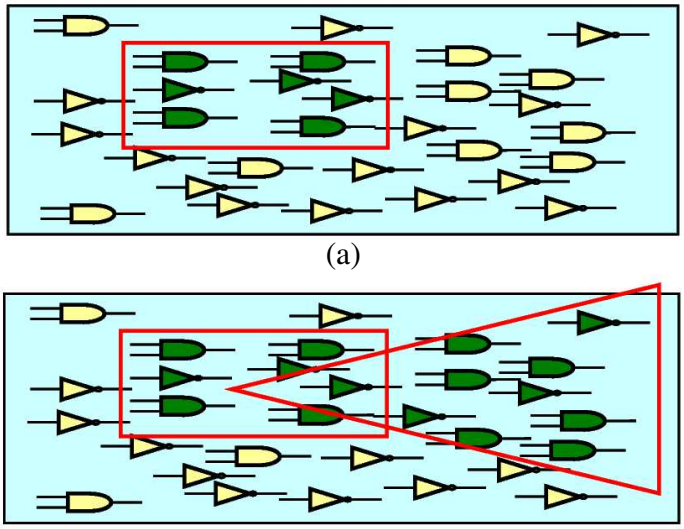


Figure 1: Similarity factor example. Note that the signatures in the fanout cone of the corrupted signal are different.



(a)

(b)

Figure 2: Resynthesis examples: (a) the gates in the rectangle are resynthesized correctly, and only their signatures may be different from the original netlist; (b) an error is introduced during resynthesis, leading to potential signature changes in the fanout cone of the resynthesized region, significantly increasing the difference factor.

Intuitively, the similarity factor of two identical circuits should be 100%. If a circuit is changed slightly but is still mostly equivalent to the original version, then its similarity factor should drop only slightly. For example, Figure 2(a) shows a netlist where a region of gates is resynthesized correctly. Since only the signatures in that region will be affected, the similarity factor is still high. 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. As Figure 2(b) shows, when a bug is introduced by resynthesis, the signatures in the output cone of the resynthesized region are also different, causing a larger drop in similarity factor. 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.

One issue that may affect the accuracy of the similarity factor is that two different signals may have identical signatures by coincidence. To counter this, a larger number of simulation vectors can be used, and the vectors can be selected carefully. For example, input vectors generated by ATPG tools typically have better signal distinguishing capabilities than those generated by random simulation. An orthogonal, but more effective technique hashes not just signatures but also the input supports of signals — two signatures match only when their input supports are also the same. Additionally, we could consider the distances from inputs to the signals in hops (depth). If two signals are found at very different distances from the inputs, they are considered different even when their signatures are identical. On the other hand, it is important to note that if one signature is wrong, its entire fanout will usually be wrong, and the chances of the entire fanout cone matching existing signatures *by coincidence* are low. This phenomenon makes the similarity factor more accurate, even without extensions for support and signal depth, for circuits with deeper logic. These extensions will primarily be useful for shallow circuits or signatures that are close to outputs, and also in the cases when we seek to precisely locate the bug at the "tip" of the incorrect fanout cone.

3.2 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 at which a signature is generated, the generated signatures should still be identical. Figure 3 shows a retiming example from [2], 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. This phenomenon becomes more obvious when the circuit is unrolled, as shown in Figure 4. Since the maximum absolute lag in this example is 1, retiming only affects gates in the first and the last cycles, leaving the rest of the circuit unmodified. As a result, signatures generated by the unaffected gates should not change.

Based on the observation above, we extend our similarity factor to sequential verification, called *sequential similarity factor*, as follows. Assume two netlists, ckt_1 and ckt_2, where the total number of signals (wires) in both circuits is N. After simulating C cycles, $N \times C$ signatures

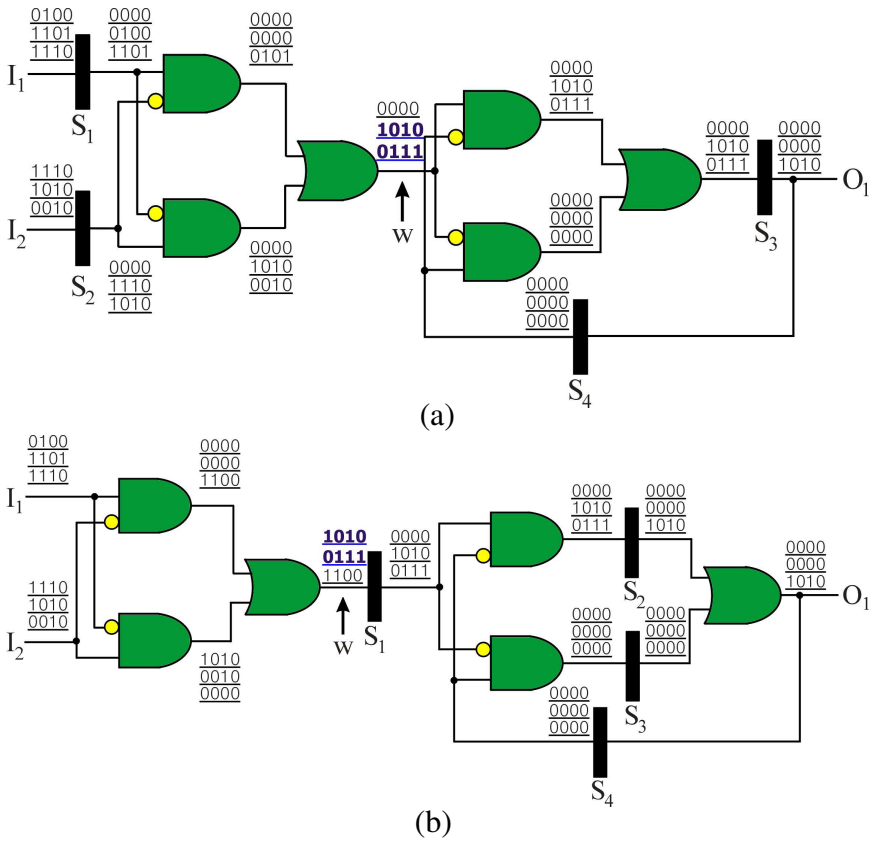


Figure 3: A retiming example: (a) is the original circuit, and (b) is its retimed version. The tables above the wires show their signatures, where the n^{th} row is for the n^{th} cycle. Four traces are used to generate the signatures, producing four bits per signature. Registers, initialized to 0, are represented by black rectangles. As wire w shows, retiming may change the cycle in which signatures appear, but it does not change the signatures themselves. Corresponding signatures are highlighted in blue (boldface).

will be generated. Among those we count M *matching* signatures. The sequential similarity factor between ckt_1 and ckt_2 is then $M/(N \times C)$. In other words:

$$sequential\ similarity\ factor = \frac{number\ of\ matching\ signatures\ for\ all\ cycles}{total\ number\ of\ signatures\ for\ all\ cycles} \qquad (2)$$

3.3 Error Visualization

The computation of similarity factor works by matching signals from two revisions of a design. Naturally, this process also identifies those nets which are unmatched to the previous design version. Those nets are responsible for the change in circuit behavior. We use two techniques to highlight the differing nets, in both cases we present the results to the user using a familiar layout format. Our first technique uses a highlight color for the gates whose output signals have unmatched signatures. In presence of an error the layout will highlight the entire output

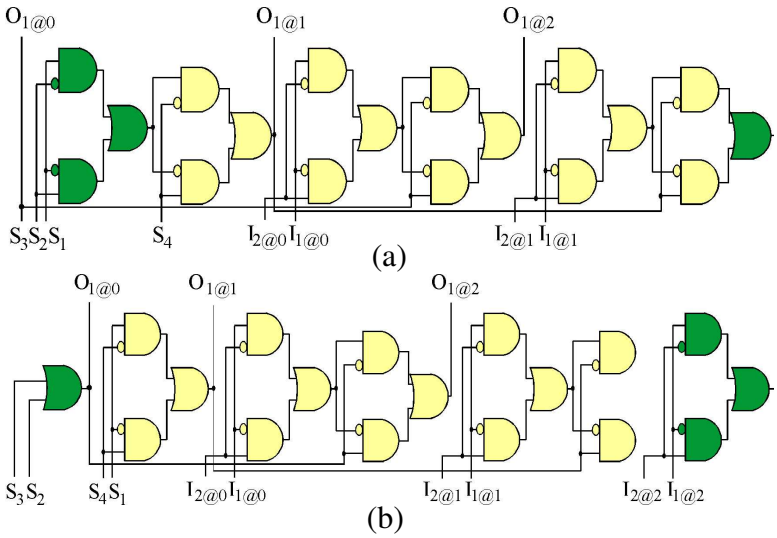
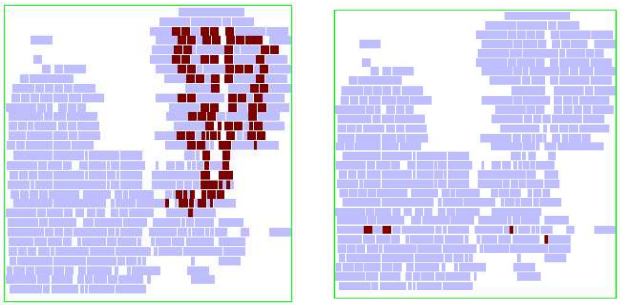


Figure 4: Circuits in Figure 3 unrolled three times. The cycle at which a signal appears is denoted using subscript "@". Retiming affects gates in the first and the last cycles (dark green), while the other gates are structurally identical (light yellow). Therefore, only the signatures of the green gates will be different.

cone of logic of the unmatched signature. The second technique only highlights the source of a problem, by marking only those gates with matched input signatures and unmatched output signatures. Figure 5 shows an example of our visualization techniques. Notice, however, that fixing these bugs may unmask other bugs, for instance in Figure 5(b) we could only detect 4 of the 5 injected bugs. The two visualization techniques described enable designers to quickly narrow down errors and find their original cause, hopefully simplifying the correlation to the synthesis optimization that generated them.

3.4 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 specific change that introduces the bug because hundreds or even thousands of changes have been processed since the last check. The similarity factor addresses this problem by pointing out the changes that might have corrupted the circuit. As described in the previous subsections, a change that greatly affects the circuit's function will probably cause a steep drop in the similarity factor. By monitoring changes in similarity factor after each circuit modification, engineers can isolate when a bug might have been introduced and trigger full equivalence checking immediately. Based on the techniques that we developed, we propose the InVerS verification methodology as follows (see Figure 6):



(a) (b)

Figure 5: 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.

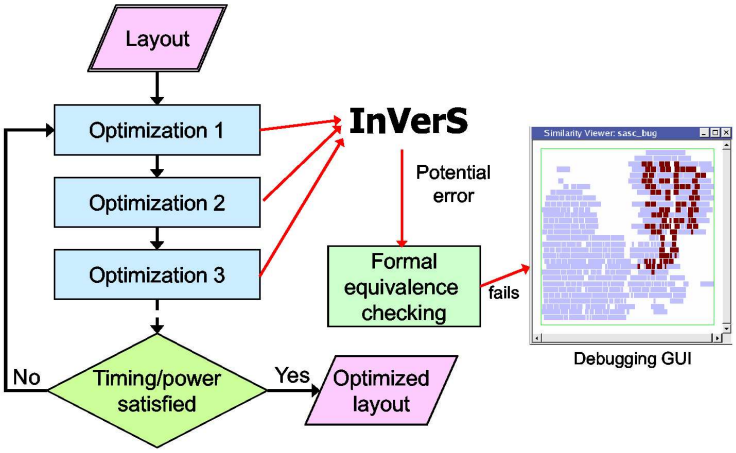


Figure 6: The InVerS verification methodology monitors each layout optimization to identify potential errors and calls equivalence checking when necessary. A debugging support GUI is provided when verification fails.

1. After each modification 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 most recent similarity factor is significantly lower. Empirically, we have found that if the latest similarity factor drops below the average by more than two standard deviations, then it is likely that the change had introduced a bug. This value, however, may vary among different benchmarks and should be empirically determined.

2. When the 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 high-

lighting the gates producing differing signals.

Since InVerS monitors drops in similarity factors, rather than absolute values of similarity factors, the structures of the netlists become less relevant. Therefore InVerS can be applied to a variety of netlists, potentially with different error-flagging thresholds. As Section 4 shows, the similarity factor exhibits high accuracy for various practical designs and allows our verification methodology to achieve significant speed-up over traditional techniques.

4 Experimental Results

We implemented InVerS using OpenAccess 2.2 and OAGear 0.96 [10]. Our testcases are selected from IWLS'05 benchmarks based on designs from ISCAS'89 and OpenCores suites, whose characteristics are summarized in Table 1. In the table, the average logic depth is calculated by averaging the logic level of 30 randomly selected gates. The logic depth can be used as an indication of the circuit's complexity. We conducted all our experiments on an AMD Opteron 880 Linux workstation. The resynthesis package used in our experiments is ABC from UC Berkeley [8]. In this section we report results on combinational and sequential verification, respectively.

Benchmark	Cell count	Ave. logic depth	Function
S1196	483	6.8	ISCAS'89
USB_PHY	546	4.7	USB 1.1 PHY
SASC	549	3.7	Simple asynchronous serial controller
S1494	643	6.5	ISCAS'89
I2C	1142	5.5	I2C master controller
DES_AREA	3132	15.1	DES cipher (area optimized)
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 (performance optimized)

Table 1: Benchmark characteristics.

Verification for combinational optimizations: in our first experiment, we perform two types of circuit modifications to evaluate the effectiveness of the similarity factor for combinational verification. In the first type, we randomly inject an error into the circuit according to Abadir's error model [1], which includes errors that occur frequently in gate-level netlists. This mimics

the situation where a bug has been introduced by an optimization. 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 [8]. 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 modified. We then use random simulation to generate 1024 patterns and calculate the similarity factor after each circuit modification for both types. Thirty samples were used 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 are anomalies, we also calculate d_2 using only SD_r. As d_2 shows, for most benchmarks the mean similarity factor drops more than two standard deviations when an error is injected. This result shows that the similarity factor is effective in predicting whether a bug has been introduced by an optimization. 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.

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

Table 2: Statistics of similarity factors for different types of circuit modifications. In this experiment we perform thirty test sets per benchmark and show the mean, minimal value (Min), maximum value (Max), and standard deviation (SD) in each row. 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.

To evaluate the effectiveness of our incremental verification methodology described in Section 3.4, we assume that there is 1 bug per 100 circuit modifications, and then we calculate the accuracy of our methodology by measuring the fraction of cases in which the similarity factor

correctly predicted equivalence. 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 more than 100X speed-up.

Benchmark	Cell count	Accuracy	Runtime(sec)	
			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. Runtimes for similarity-factor (SF) and equivalence checking (EC) are also shown.

Sequential verification for retiming: in our second experiment, we implement the retiming algorithm described in [6] and use our verification methodology to check the correctness of our implementation. This methodology successfully identified several bugs in our implementation. In our experience, most bugs were caused by incorrect netlist modifications when repositioning the registers, and a few bugs were due to erroneous initial state calculation. Examples of the bugs include: (1) incorrect fanout connection when inserting a register to a wire which already has a register; (2) missing/additional register; (3) missing wire when a register drives a primary output; and (4) incorrect state calculation when two or more registers are connected in a row.

To quantitatively evaluate our verification methodology, we ran each benchmark using the correct implementation and the buggy version to calculate their respective sequential similarity factors, where 10 cycles were simulated. The results are summarized in Table 4, which shows that the sequential similarity factors for retimed circuits are 100% for most benchmarks. As ex-

plained in Section 3.2, only a few signatures should be affected by retiming. Therefore, the drop in similarity factor should be very small, making sequential similarity factor especially accurate for verifying the correctness of retiming. This phenomenon can also be observed from Table 5, where the accuracy of our verification methodology is higher than 99% for most benchmarks. To compare our methodology with formal equivalence checking, we also show the runtime of a sequential equivalence checker based on bounded-model-checking in Table 5. This result shows that our methodology is more beneficial for sequential verification than combinational because sequential equivalence checking requires much more runtime than combinational. Since the runtime to compute sequential similarity factor remains small, our technique can still be applied after every retiming optimization thus eliminating most unnecessary sequential equivalence checking calls.

Benchmark	Sequential similarity factor (%)							
	Retiming without errors				Retiming with errors			
	$Mean_r$	Min_r	Max_r	SD_r	$Mean_e$	Min_e	Max_e	SD_e
S1196	100.0000	100.0000	100.0000	0.0000	98.3631	86.7901	100.0000	3.0271
USB_PHY	100.0000	100.0000	100.0000	0.0000	99.9852	99.6441	100.0000	0.0664
SASC	99.9399	99.7433	100.0000	0.0717	99.9470	99.3812	100.0000	0.1305
S1494	100.0000	100.0000	100.0000	0.0000	99.0518	94.8166	99.5414	1.5548
I2C	100.0000	100.0000	100.0000	0.0000	99.9545	99.6568	100.0000	0.1074
DES_AREA	100.0000	100.0000	100.0000	0.0000	95.9460	69.1441	100.0000	6.3899

Table 4: Statistics of sequential similarity factors for retiming with and without errors. In this experiment we perform thirty test sets per benchmark and show the mean, minimal value (Min), maximum value (Max), and standard deviation (SD) in each row.

Benchmark	Cell count	DFF count	Accuracy	Runtime (sec)	
				SEC	SSF
S1196	483	18	99.87%	5.12	0.42
USB_PHY	546	98	99.10%	0.41	0.34
SASC	549	117	95.80%	5.16	0.56
S1494	643	6	99.47%	2.86	0.45
I2C	1142	128	99.27%	2491.01	1.43
DES_AREA	3132	64	99.97%	49382.20	14.50

Table 5: Runtime of sequential similarity factor calculation (SSF) and sequential equivalence checking (SEC). The accuracy of our verification methodology is also reported, where 1 bug per 100 retiming optimizations is assumed.

5 Conclusions

In this work we developed a novel incremental verification and debugging 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 fast simulation, which can efficiently calculate a "similarity factor" metric to spot potential differences between two versions of a design. The areas 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.

References

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

[2] K.-H. Chang, D. A. Papa, I. L. Markov and V. Bertacco, "InVerS: An Incremental Verification System with Circuit Similarity Metrics and Error Visualization", *ISQED'07*, pp.487-492.

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

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

[5] 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.

[6] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry", *Algorithmica*, 1991, Vol. 6, pp. 5-35.

[7] Q. Zhu, N. Kitchen, A. Kuehlmann and A. Sangiovanni-Vincentelli, "SAT Sweeping with Local Observability Don't-Cares", *DAC'06*, pp. 229-234.

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

[9] "Conformal finds DC/PhysOpt was missing 40 DFFs!", ESNUG 464 Item 4, Mar. 30, 2007.

[10] http://www.si2.org/

[11] "Future of Chip Design Revealed at ISPD", EE Times, Apr. 17, 2008.