Fast Verification of Retiming

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

ABSTRACT

Retiming is a powerful logic optimization technique that repositions registers in a circuit. However, its verification is difficult. In this work we implement a classical retiming algorithm and check it using a sequential verification methodology that evaluates the correctness of retiming using fast simulation. Unlike traditional verification techniques that are demanding in memory and computing power, this methodology quickly discovers and isolates most errors caused by retiming, thus reducing verification effort.

1. INTRODUCTION

In order to satisfy required performance and power constraints in modern circuits, it is now common to apply aggressive optimizations, such as retiming. Unlike combinational logic optimizations, retiming changes the number and locations of registers to optimize various objectives such as clock period, area and power. Verifying the correctness of retiming, however, is much harder than combinational equivalence checking. Given that subtle and unexpected bugs still appear in physical synthesis tools today [4], the complexity of retiming verification greatly limits its practical use.

In this work we implement a retiming algorithm based on [2] to enrich the logic optimization capability of OAGear which currently lacks this functionality. In addition, we propose a fast sequential verification methodology to verify the correctness of retiming. To this end, we define a *Sequential Similarity Factor (SSF)* that can quickly estimate the correctness of retiming using simulation. We also implement a formal equivalence checker based on Bounded Model Checking (BMC) to verify these estimates. Our empirical results show that SSF calculation can be up to three orders of magnitude faster than formal methods and requires considerably less memory. Therefore, it can be applied much more often during physical synthesis to localize any errors introduced by logic transformations, thus facilitating error isolation and debugging.

The rest of the paper is organized as follows. In Section 2 we outline our retiming implementation and initial state calculation. Our verification methodology is described in Section 3. Empirical results are shown in Section 4, and Section 5 concludes this paper.

2. RETIMING

Retiming is a sequential circuit optimization technique that repositions the registers while leaving the combinational cells untouched. It can be used to optimize various objectives, such as the clock period or register count in a circuit. The functional behavior of the circuit, however, is still preserved, as Figure 1 illustrates. In this section, we first describe our implementation of retiming, and then outline how new initial states are calculated for the retimed circuit.

2.1 Implementation

In our retiming package we implemented algorithm OPT1 described in [2] that optimizes clock period. The inputs to our package include a mapped netlist and the delays of its combinational cells. The output of the package is a retimed netlist whose clock period is minimized. If the delays for the cells are not given, unit delay is assumed. Our implementation utilizes the SimEqui, Ai, Func and Util packages. We followed the OAGear programming guidelines, provided comprehensive documentation, and included a regression testing environment.

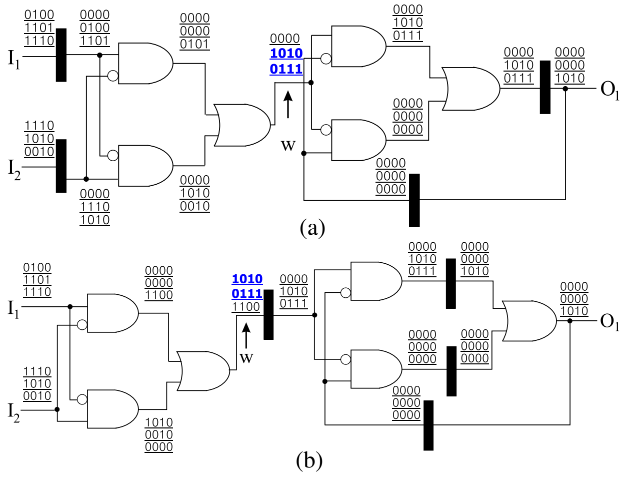

Figure 1: A retiming example: (a) is the original circuit, and (b) is its retimed version. The tables above the wires show their signatures, where the n^{th} row is for the n^{th} cycle. Four traces are used to generate the signatures, producing four bits per signature. Registers, initialized to 0, are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

2.2 Initial State Calculation

After a circuit is retimed, its initial state may be different due to the change in register locations. While the new states for forward-retimed registers (lag < 0) can be calculated easily via simulation, the computation of new states for backward-retimed registers (lag > 0) is unfortunately an NP-complete problem [3]. In our implementation, we use SAT to find new values for backward-retimed registers. Currently, we support the calculation of new states for unlimited negative lag, but we restrict the maximum positive lag to 1. The user is allowed to specify the initial state of the original circuit for new state calculation. If this information is not available, it is assumed that all the registers are initialized to 0.

3. VERIFICATION OF RETIMING

To reliably implement retiming, it is necessary to have a verification methodology that can quickly identify most problems and point out their sources. In this section, we first propose the concept of *Sequential Similarity Factor (SSF)* that can predict the sequential equivalence between a netlist and its retimed variant with high accuracy. Next, we describe a formal equivalence checker based on Bounded Model Checking (BMC) and illustrate our overall verification methodology for retiming.

3.1 Sequential Similarity Factor

The SSF between two sequential circuits is defined 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 will be generated, where a k-bit signature is a sequence holding the simulation values on each of the k input patterns. Out of those signatures, we distinguish M *matching* signatures — a signature is considered matching if and only if both circuits include

Benchmark	Gate count	DFF count	Retiming		Verification (k=1024)					($k = 10240$)
			Delay improv.	Runtime (sec)	Mean SSF after retiming		Runtime (sec)			Mean SSF
					Without errors	With errors	SSF	SEC	Speed-up	with errors
s1196	483	18	**0.00%**	4.6	100.0000%	98.3631%	0.4	5.1	**12.7×**	97.6040%
usb_phy	546	98	**11.90%**	4.1	100.0000%	99.9852%	0.3	0.4	**1.3×**	99.9852%
sasc	549	117	**16.95%**	3.8	99.9399%	99.9470%	0.6	5.2	**8.7×**	99.9470%
s1494	643	6	**7.28%**	11.1	100.0000%	99.0518%	0.5	2.9	**5.8×**	99.0342%
i2c	1142	128	**12.50%**	44.9	100.0000%	99.9545%	1.4	2491.0	**1779.3×**	99.9485%
des_area	3132	64	**0.17%**	4294.5	100.0000%	95.9460%	14.5	49382.2	**3405.7×**	95.8700%

Table 1: Circuit delay improvement due to retiming, and results of our verification methodology: mean sequential similarity factors (SSFs) for retimed circuits with and without errors. Runtime is shown for retiming, SSF calculation and sequential equivalence checking (SEC). The speed of our verification methodology allows it to be applied frequently to facilitate debugging. In addition, SSF uses significantly smaller amount of memory than SEC (16M vs. 1386M for des_area). In the table, k is the number of patterns per cycle used for SSF calculation. Larger k causes larger drop in SSF when errors are introduced, which improves the accuracy of SSF.

the signature (this can be found efficiently by hashing). The SSF between ckt_1 and ckt_2 is then $M/(N \times C)$. In other words:

$$SSF = \frac{number\ of\ matching\ signatures\ for\ all\ cycles}{total\ number\ of\ signatures\ for\ all\ cycles} \qquad (1)$$

The intuition behind this definition is that retiming only changes the register positions but leaves combinational gates intact. As a result, the signatures of the retimed circuit should mostly remain the same although they may appear at different cycles, as can be observed from Figure 1 [1]. By considering multiple cycles in SSF, we can capture most identical signatures that appear at different cycles. When errors are introduced by a retiming operation, signatures that only exist in the new netlist will appear, which will cause a drop in SSF between the original and the retimed netlists.

3.2 BMC-based Equivalence Checking

To formally validate the sequential equivalence between two circuits and to evaluate our SSF methodology, we implemented a BMC-based checker that verifies equivalence up to C cycles. Given two circuits, we first unroll both circuits C times. Next, we connect the primary inputs of both circuits for each unrolled copy and constrain the circuits using their initial states. We then add miters to the primary outputs between two circuits, feed their outputs to an OR gate, and set the output of the gate to 1. The circuit is converted to a CNF and solved by a SAT solver. If the CNF is unsatisfiable, then the circuits are equivalent up to C cycles; otherwise, a counterexample will be returned to show their discrepancy. Although this technique can only check equivalence up to C cycles, it should be able to catch most bugs introduced during retiming.

3.3 Overall Verification Methodology

Our verification methodology works as follows: after each retiming operation, we calculate the SSF between the original and the retimed netlist. If SSF drops by more than 2 standard deviations (calculated using the running average/variance of the past 30 SSFs; the threshold is determined empirically), we call sequential equivalence checking. If verification fails, we return a counterexample to the user for debugging; otherwise, it is highly likely that the two netlists are equivalent, and the engineer can decide whether more comprehensive sequential verification should be conducted.

4. EXPERIMENTAL RESULTS

To evaluate our retiming package, we conducted experiments using IWLS'05 benchmarks on an AMD Opteron 880 Linux workstation. We assigned random delay to each gate to generate various retiming configurations, and we ran each benchmark 30 times. The delay improvement and runtime of retiming are summarized in Table 1. In addition, we used our verification methodology to check the correctness of our retiming implementation, and this methodology successfully identified several subtle bugs. In our experience, most bugs were caused by rare cases of incorrect netlist transformations when repositioning the registers, and a few bugs were due to erroneous initial state calculation. Such bugs include: (1) incorrect fanout connection, when inserting a register, to a wire which already has a register; (2) missing or spurious registers; (3) missing wire when a register drives a primary output; and (4) incorrect state calculation when two ore more registers are connected in a row.

To quantitatively evaluate our verification methodology, we retimed each circuit using the correct and the buggy implementations of OPT1. Then we used 10 cycles of simulation to calculate mean SSFs in these two cases. In practice, one can use many more cycles of simulation to refine signatures as well as several different simulation traces. We also show the runtime of Sequential Equivalence Checking (SEC) for comparison, where the depth was also set to 10 cycles. The results are summarized in Table 1, which show that the average SSF is much lower in erroneous circuits than correctly-retimed circuits for most benchmarks. This phenomenon suggests that SSF can effectively predict whether or not a bug has been introduced. The runtime comparison between the calculation of SSF and SEC shows that calculating SSF can be orders of magnitude faster than SEC, especially for larger designs. As a result, it can be applied frequently to discover and isolate bugs when they first occur. Additionally, we validated the SSF calculation on circuits with up to 98341 cells and 8808 flip-flops, and observed linear scaling of runtime, as expected.

Note that our SSF methodology works directly with optimized circuits and can verify a variety of optimization techniques, not only the OPT1 retiming algorithm. We believe that it scales better than the majority of formal methods, not only the naive BMC variant we have implemented. However, OAGear does not currently offer such techniques, which makes comparisons difficult.

5. CONCLUSIONS

Retiming is a powerful logic optimization technique that is gaining popularity in the industry, but it is currently missing in OAGear. Our work contributes an implementation of retiming that enriches OAGear's logic optimization capability. Our second contribution is a sequential verification methodology that was used to check this implementation — it is fast and accurate enough to be used after every retiming step rather than as an expensive post-processor after all optimizations.

6. REFERENCES

[1] 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.

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

[3] L. Stok, I. Spillinger, and G. Even, "Improving Initialization through Reversed Retiming", *EDTC'95*, pp. 150-154.

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