Minimum-Perturbation Retiming for Delay Optimization

Sayak Ray Alan Mishchenko Robert Brayton
Department of EECS
University of California, Berkeley
{sayak, alanmi, brayton}@eecs.berkeley.edu

Stephen Jang
LogicMill Technology
San Jose, CA
sjang@logic-mill.com

Thomas Daniel
Abound Logic
Santa Clara, CA
tdaniel@aboundlogic.com

ABSTRACT

This paper describes a fast retiming algorithm targeting delay while minimizing the number of flip-flops moved. The algorithm can be applied before placement to minimize logic level, or after placement to minimize the critical region annotated with wire delays. Experiments on a suite of industrial benchmarks show that the algorithm improves fMAX by 9% while increasing LUT count by 1% and flip-flop count by 2%, respectively. The runtime penalty is less than 1% of the total runtime of the design flow.

1. INTRODUCTION

Given a synchronous design, retiming [6] moves sequential elements, or flip-flops (FFs), over combinational nodes, to reduce delay (the longest combinational path), area (the total FF count), power consumption, and/or other metrics.

Retiming plays an important role in the arsenal of methods used to optimize sequential circuits. In addition to targeting area and delay, numerous publications over the last two decades addressed various aspects of retiming: initialization [14], incrementality [12][15], integration with logic restructuring [10], technology mapping [8], and both [7], to mention just a few.

However, we are not aware of any published work on retiming that focuses on minimizing the number of FFs moved to achieve a given objective (for example, a delay target).

Our main contribution in this work is a new minimum perturbation approach to retiming for delay optimization. The approach is useful in technology independent synthesis, to minimize the level count, and in post-placement resynthesis when (approximate) wire delays are known, to reduce the clock cycle.

To use retiming in post-placement resynthesis, the placement tool must perform incremental placement, i.e. only re-place the modified elements and retain the original placement as much as possible. If the placement tool is not capable of incremental placement, the placement-resynthesis loop may not converge, since each new placement creates drastically different wire delays.

Furthermore, when retiming is used in the post-placement scenario, it is often important that changes to the design remain as small as possible. Otherwise, the predicted delay improvement due to retiming may not be realized because the backend tool failed to come up with an efficient incremental placement.

We implemented our retiming solution for arbitrary sequential logic networks with multiple clock domains and FFs with complex controls and setup/hold constraints. The algorithm assumes that the delays of combinational nodes and wires, given at the beginning, are preserved after retiming, and only the FFs move from the output of one logic node to the output of another. This assumption holds for FPGAs, in which each LUT can be programmed to implement an FF at its output. We tested our tool as part of a complete FPGA design flow [1] and report promising results on a suite of industrial designs.

The approach of this paper is similar to incremental physical retiming in [12], in that it does not use linear programming [6] and, therefore, produces valid intermediate results and handles complex architectural constraints (such as adder chains).

The main difference, in addition to several important details, is that [12] tries to move *all* critical FFs at a time, while our methods moves only *one* critical FF at a time. This gives our method the following advantages:

- For the unit delay model, our algorithm is exact in that it achieves the desired delay by retiming the provably minimum number of flops. (See Section 4.)
- The evaluation and realization phases of our algorithm are separated, making it faster than [12], as implemented by us in ABC [2]. (See the experimental results.)

The proposed approach is also similar to [4]; the trade-off between the number of FF moved and the area/delay measurement achieved is explored. However, [4] uses heuristic delay-optimization similar to [12], which may be suboptimal and relatively slow for large designs, compared to our method.

The rest of this paper is organized as follows. Section 2 describes the background. Section 3 describes the algorithm. Section 4 outlines the experimental environment. Section 5 reports experimental results. Section 6 concludes the paper and outlines future work.

2. BACKGROUND

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates or lookup-tables (LUTs) and directed edges corresponding to connecting wires. The terms Boolean network, netlist, and design are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D-*flip-flops* (FFs) with initial states. Retiming can only be applied to a sequential logic network.

FFs may have *complex controls* (set, reset, enable, etc), but most computations including retiming treat them as single-input, single-output sequential boxes while translating the FF controls into additional nodes, as shown in [3]. For example, an enable signal becomes a MUX controlled by the enable, with the *then*-edge fed by the FF data-input, and the *else*-edge by the FF output.

A node n has zero or more *fanins,* i.e. nodes that are driving n, and zero or more *fanouts*, i.e. nodes driven by n. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a designated subset of nodes of the network. A *fanin* (*fanout*) *cone* of node n is a subset of all nodes of the network, reachable through the fanin (fanout) edges of n.

A *node* is a logic component having a propagation delay. An *edge*, also called *wire*, is the pin-to-pin connection between two adjacent nodes. The delay of a path includes *logic delays* and *wire delays*. The *logic delay* occurs in a logic component, such as a LUT. The *wire delay* occurs on the edges.

In modern FPGAs, the delay for each pin in a LUT is different, so a variable-pin-delay model is used in this paper. Wire delays are usually not known until placement and routing. To approximate wire delays, initially a fixed delay is added to the delay of all pins of the LUTs in the library. After placement, the wire-delays or their close approximations are known. Therefore, for the needs of the post-placement retiming discussed in this paper, we assume that the input edges of objects are annotated with numbers representing the corresponding wire delays.

Additionally, the logic network considered for retiming may contain combinational *white-boxes* and sequential *black-boxes*. A *white-box* is an object whose logic function and delay are known to the tool. Retiming can move FFs over the white-box, but it cannot leave them inside the box. An example of a white-box is an adder. A *black-box* represents an object whose logic is not known to the tool, but since this paper assumes these boxes to be sequential and clocked by the same clock as FFs, they can be handled as terminals of retiming. Examples of black-boxes are register files, memories, and dedicated arithmetic modules.

The *sequential inputs* (*outputs*) of a clock domain are outputs (inputs) of FFs and sequential boxes synchronized by the clock.

If the network contains flops with *multiple clock-domains*, each clock domain may be retimed separately, while assuming that only the flops of that domain can move; all other flops are treated as additional PIs and POs.

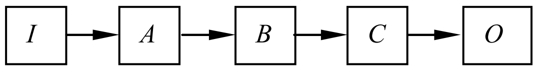
3. ALGORITHM

This section describes the proposed retiming algorithm.

3.1 Timing analysis

This subsection introduces the relevant timing terminology and computations used in the paper. We deviate from the traditional presentation in terms of arrival/required times [11]; instead we use arrival/departure times, because the computations are more intuitive when formulated this way. The relationship between the two notions is also described in this subsection.

The *arrival times* of all objects are computed by finding the longest path (using pin-to-pin delays as well as edge delays) from sequential inputs. An arrival time of a combinational object (node or white box) includes its intrinsic delay. The *departure times* of all objects are computed by finding the longest path (again using pin-to-pin delays and edge delays) to sequential outputs. A departure time of a combinational object does not include its intrinsic delay. The following schematic illustrates the concept of arrival time and departure time.



Here *I* and *O* represent a sequential input and a sequential output, respectively, while *A, B,* and *C* are combinational objects. The arrival time of object *B* along the path *I-A-B* is

$$e\_delay(I, A) + i\_delay(A) + e\_delay(A, B) + i\_delay(B)$$

where *e_delay(x, y)* is the delay of the edge between *x* and *y*, and *i_delay(x)* is the intrinsic delay of object *x*, that is, the delay between the input pin and the output pin connecting *x* to the objects on a given path. The arrival time of *B* is, therefore, the maximum of the arrival times over all paths from any sequential input to *B*. The departure time of B along the path B-C-O is

$$e\_delay(B, C) + i\_delay(C) + e\_delay(C, O).$$

The overall departure time of *B* is the maximum of departure times over all paths from *B* to any sequential output.

Slack of an object (x) is computed as follows:

$$Slack(x) = MaxDelay(\mathrm{N}) - Arrival(x) - Departure(x),$$

where *MaxDelay*(N) is the maximum combinational delay of the network, that is, delay of the longest path between any sequential output and sequential input. Intuitively, *slack of an object* is the largest delay that can be added to the arrival time of the object, without increasing the maximum delay of the network. Similarly, *slack of an edge* is the largest delay that can be added to the arrival time of the edge, without increasing the maximum delay of the network. Note that even if the slack of two nodes is zero, the slack of an edge between these nodes may be non-zero.

The *required time* of an object is the latest time for the signal to arrive at the object without increasing the maximum delay of the network. Required time can be expressed using departure time:

$$Required(x) = MaxDelay(\mathrm{N}) - Departure(x),$$

which can be substituted into the above equation for the slack to get the familiar relationship between the timing parameters:

$$Required(x) = Arrival(x) + Slack(x).$$

3.2 Incremental retiming algorithm

In this section, we introduce the retiming algorithm, which uses a simple wire-delay model. The same algorithm can be applied using a unit-delay model by assuming delays of wires connecting objects to be 0, and intrinsic delays of objects to be 1. To make delay computation more realistic for FPGA applications, the intrinsic delay between carry-in and carry-output of adders is assumed to be 0.

The same algorithm can be applied to the design after place-and-route. In this case, the network objects and edges are annotated as follows: (i) each object is assigned a coordinate in the placement, and (ii) each edge is assigned a wire-delay (given separately or derived from placement using a delay model).

The presentation in this section is limited to backward retiming, which requires incremental computation of arrival times. The case of forward retiming is easier because the computation of the initial state of the retimed FFs is straight-forward. Another difference is that departure times are used instead of arrival times.

Retiming involves three steps described below: (a) preparation of data structures, (b) evaluation of retiming possibilities, and (c) realization of retiming on the design.

3.2.1 Preparation

The data structures are populated with the following data:

- *Network* composed of PIs/POs, FFs, LUTs, boxes, etc.
- *Object delays*: mapping of each fanin of a combinational object (node or white-box) into its intrinsic delay.
- *Wire delays*: mapping of each fanin/fanout connection into an integer number.

The following user-specified parameters affect the algorithm:

- The upper bound on delay improvement (if delay improvement is more than this, stop; this parameter is useful to control area increase).
- The lower bound on delay improvement (if delay improvement is less, stop; this parameter prevents retiming moves that are unlikely to help).
- The limit on the number of retiming moves without improvement (when the limit is reached, stop and declare that no improvement is possible).
- The limit on the total number of nodes retimed.
- The limit on area increase.

To speed up an evaluation of retiming possibilities, a specialized data structure is constructed, representing FFs as temporary attributes on the edges of the original network. These attributes can be efficiently moved forward and backward without affecting positions of the real FFs. Before the evaluation begins, the arrival times of each object in the network, including the temporary FF attributes, are computed, and a priority queue is initialized to access FF attributes by their arrival times.

To simplify the presentation in the next subsection, when referring to manipulating FFs, only the temporary FF attributes are actually being manipulated. The retiming moves are recorded during evaluation in terms of the FF attributes, to be reproduced using the real FFs in the final phase of the algorithm, when the actual retiming of the real design is performed.

3.2.2 Evaluation

During this step, the following is iterated:

- The flops are examined in the queue in decreasing order of their arrival times.
- The first flop that can be retimed backward is extracted from the queue.
- The flop is retimed over its fanin node.
- The arrival times of all objects in the TFO cone of the recently retimed node are incrementally updated.
- The positions of the flops whose arrival times have changed are updated in the queue.

This iteration stops when any of the following is true:

- The delay target is reached.
- No delay improvement is observed after moving the given number of FFs.
- No FFs in the queue can be retimed.
- The queue contains no FFs.

When iteration terminates, the algorithm has found a trade-off between the delay improvement and the number of FFs moved. Figure 1 shows an example of this trade-off: moving the first 10 FFs allows a reduction in delay of one unit, while moving an additional 15 FFs allows a reduction of delay by another unit. No further FF moves have led to delay reductions.

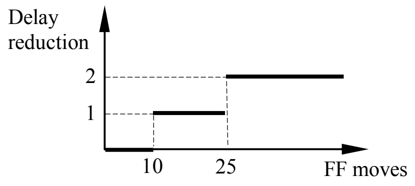


Figure 1. Illustration of delay improvement vs. FF moves.

The trade-off computed by the algorithm allows us to choose a retiming that (a) yields a required delay if reduction can be achieved, and (b) satisfies the user-specified constraints on the number of FFs moved and/or area increase.

3.2.3 Realization

During this step, the FF moves observed during evaluation are actually performed on the FFs of the design. As a result, the FFs travel from their initial positions to where they should be to obtain the best trade-off between the delay improvement and the number of FFs moved.

The following additional steps are carried out during this phase:

- If the flops to be moved have different controls, these controls are elaborated into additional logic nodes [3].
- If backward retiming was performed, the initial state is computed using a call to the SAT solver (if the initial state does not exist, retiming is undone).
- The new nodes and FFs created by retiming are assigned new names (the nodes that have been retimed over can be renamed too, if required by incremental placement).

When the final network after retiming is ready, it can be written into a file or returned to the caller via programmable APIs. Optionally, the new network can be verified against the original one by sequential simulation or sequential equivalence checking.

4. Theoretical considerations

In this section, we discuss the main theoretical claim of this paper. We prove that our algorithm will move the minimum number of flops to achieve a target delay under the unit-delay model, assuming backward-only retiming. A sketch of proof is presented here. For the unit-delay model, our algorithm is similar to [15] where an elaborate proof of optimality can be found.

Proof (Sketch): This proof holds for the simplistic graph model of a network where a network is viewed as a directed graph of AND nodes and D-flip-flops with no combinational cycles. A network with delay d in this model contains one or more critical paths of length d where a path is critical iff all nodes along it have slack zero. Under this model, our algorithm is similar to [15] but differs in its objective. While [15] continues to iterate retiming moves until it achieves the minimum delay, our algorithm stops once it reaches any of the resource limits. We aim at a minimum delay retiming under the perturbation constraint.

Suppose p is a critical path of length d, and *final(p)* is the final node of p. To reduce delay of the network by backward retiming, we need to break every critical path p by moving a flop over its *final(p)* from its fanout edge(s) to its fanin edge(s). Note that we may expose new critical paths while breaking existing ones and we need to break them all by backward retiming. Hence, if there are n nodes serving as *final(p)* for a set of critical paths $\{p\}$ (either existing or new), we need to perform exactly n rounds of backward retiming, unless the given delay is the optimum one.

Note that, this scheme is order-independent; it does not matter which critical path is broken first, since all critical paths need to be broken. This is what our algorithm does and therefore optimality is proved for the unit-delay model. □

Note that this proof holds for a simple graph model of the network. We are investigating similar optimality claims for networks with multi-output boxes and other complex components, as well delay models that are other than unit-delay model.

5. EXPERIMENTAL ENVIRONMENT

Post-placement retiming (pRetime) was tested using the Raptor Development Environment (RDE) from Abound Logic. RDE is a complete FPGA design flow targeting Raptor devices.

The main features of RDE are summarized below. For a more detailed description, refer to [1].

The flow starts by RTL synthesis and is followed by network mapping. Then global placement, detail placement, routing, and timing analysis are performed. The placer can perform incremental placement by automatically detecting netlist differences, detecting instances whose placement did not change, and placing only the new instances. Some of the old instances can be re-placed to allow for a better placement of the new instances.

A Tcl-based user interface from RDE can be used to customize of the entire flow. Experiments for this paper were performed using N iterations between incremental placement and pRetime. The iteration count, N, is a command line option of the script.

Initially, mapping, global placement, and detailed placement are performed once. The timing score for the placed network is provided by the detailed placer. During each iteration, from 1 to N, the following steps are executed:

- run pRetime using one of the predefined strategies;
- run incremental placement followed by the computation of a new timing score;
- the optimized netlist is accepted, if the new score is better.

6. EXPERIMENTAL RESULTS

6.1.1 Evalution as part of the flow

This experiment was performed using 20 large industrial designs ranging from 206K to 678K LUT4. The results for these designs are shown in Table 6.1. The table contains two runs:

- Section "Baseline" is a typical RDE flow without retiming.
- Section "pRetime" is the same RDE flow with 10 iterations of post-placement retiming.

Columns "LUT" and "FF" shows the LUT count and the register count, respectively. Column "fMAX" shows the maximum frequency, computed using static timing analysis. Column "Time" shows the total runtime excluding RTL synthesis. Row "Geomean" shows the geometric mean for the specific column. Row ratio shows the ratio of "pRetime" vs. "Baseline" for the given metric.

The following observations can be made about the experiments.

- fMAX is improved by 9.7%.
- LUT count is degraded by only 1.0%.
- FF count is degraded by only 2.0%.
- The runtime penalty is only 0.8%.

6.1.2 Comparison with previous work

In this experiment, we compared the proposed retiming with the incremental delay-oriented retiming [12], as implemented in ABC (command *retime –M 4 –b*).

The experiment was performed using four large industrial designs. The results are shown in Table 6.2. Section "Profile" lists the LUT count, the FF count, and the level count for each design. The following sections show the number of levels reached by the retiming algorithm, and the runtime needed for that.

The two algorithms are similar in the way they move FFs to reduce the length of the longest combinational path, but they differ in the way they choose what flops to move. The algorithm of [12] moves *all* critical FFs that can be moved, while the proposed algorithm always moves *one* most critical FF among those that can be moved. As a result, the proposed algorithm determines the need for retiming more accurately.

It should be noted that comparing the two algorithms was hard because they are implemented using different data-structures, in particular, the data-structure for [12] in ABC does not support white-boxes. Therefore, to make a fair comparison, we collapsed the white-boxes and re-mapped the designs into 4LUTs. Another problem was that the legalization strategies used after retiming are different, which makes the number of flops incomparable, although we observed that the FF count after the proposed algorithm tends to be smaller.

7. CONCLUSIONS

This paper introduced a new retiming algorithm that can be used during both technology-independent optimization and after placement. The algorithm has the following salient features:

- The user can explore the range of feasible delay solutions.
- The minimum perturbation for a delay target is guaranteed for the unit-delay model and single-output objects.
- Substantial delay reductions on industrial designs are often achieved (fMAX is reduced by 9.7% in our experiments).
- The algorithm is fast (takes only a few seconds for designs with millions of nodes and hundreds of thousands of FFs).

Future work will include:

- Experiments with unit-delay retiming and understanding its impact on the delay after place-and-route.
- Investigating why a predicted delay improvement does not always match the actual delay improvement.
- Understanding why forward retiming is rarely useful.
- Developing a more general incremental algorithm which alternates forward and backward retiming.
- Exploring the possibility of integrating retiming with logic structuring and technology mapping [7][8].
- Providing a more formal proof of the theoretical claims.
- Experimenting with the optimality of the algorithm for various delay models.

8. ACKNOWLEDGMENTS

This work is supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668, and the industrial sponsors Abound Logic, Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.

9. REFERENCES

[1] Abound Logic. *Raptor Development Environment*. Product brief. PB003 (V1.0), Dec 2009. http://www.aboundlogic.com/

[2] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. Release 00127p. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[3] K. Eckl, J.-C. Madre, P. Zepter, and C. Legl. "A practical approach to multiple-class retiming". Proc. DAC'99, pp. 237-242.

[4] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Minimizing implementation costs with end-to-end retiming", *Proc. IWLS '07*, pp. 9-16.

[5] A. P. Hurst, A. Mishchenko, and R. K. Brayton. "Scalable min-area retiming under simultaneous delay and initial state constraints". *Proc. DAC'08*, pp. 534-539.

[6] C. E. Leiserson and J. B. Saxe. "Retiming synchronous circuitry". *Algorithmica*, 1991, Vol. 6(1), pp. 5-35.

[7] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan. "Integrating logic synthesis, technology mapping, and retiming", *ERL Technical Report*, EECS Dept., UC Berkeley, April 2006.

[8] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton. "Combinational and sequential mapping with priority cuts", *Proc. ICCAD '07*, pp. 354-361.

[9] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," *Proc. FPGA '98*, pp. 35-42.

[10] P. Pan. "Performance-driven integration of retiming and resynthesis". *Proc. DAC'99*, pp. 243-246.

[11] S. Sapatnekar. "Timing", Springer Verlag, 2004

[12] D. P. Singh, V. Manohararajah, and S. D. Brown. "Incremental retiming for FPGA physical synthesis". *Proc. DAC'05*, pp. 433-438.

[13] P. Suaris, D. Wang, and N.-C. Chou, "A practical cut-based physical retiming algorithm for field programmable gate arrays". Proc. ASPDAC'05, pp. 1027-1030.

[14] H. J. Touati and R. K. Brayton. "Computing the initial states of retimed circuits", *IEEE TCAD,* 1993, Vol. 12(1), pp. 157-162.

[15] J. Wang and H. Zhou. "An efficient incremental algorithm for min-area retiming". *Proc. DAC'08*, pp. 528-533.

Table 6.1. Experimental evaluation of retiming after placement on industrial circuits.

Designs	**Baseline**				**pRetime**			
	LUT	**FF**	**fMAX (MHz)**	**Time (hours)**	**LUT**	**FF**	**fMAX (MHz)**	**Time (hours)**
D01	206725	132416	105.37	1.313	206725	132416	115.34	1.221
D02	212124	64120	68.82	0.647	212621	64463	67.61	0.646
D03	216810	8581	93.28	0.327	216810	8581	95.15	0.312
D04	296152	133704	89.93	0.721	298878	134123	92.25	0.889
D05	323818	86712	40.68	1.994	326869	89150	41.19	1.972
D06	370429	105650	59.88	1.289	371745	106674	63.57	1.559
D07	413017	195150	81.50	1.399	413017	195150	81.50	1.302
D08	416309	223992	147.28	2.157	416432	224127	150.83	2.090
D09	455429	160450	27.53	2.234	478726	182627	51.52	1.875
D10	470436	230811	53.59	3.304	472466	231477	55.74	3.795
D11	519766	186258	27.56	2.762	559205	219243	48.08	2.746
D12	522988	311436	68.78	1.834	523483	311546	69.11	2.010
D13	523203	311641	69.06	1.802	524324	312111	69.64	1.891
D14	575355	351911	136.05	2.588	575355	351911	136.05	2.434
D15	599413	216051	202.02	1.069	599413	216051	209.64	1.096
D16	618377	259844	47.66	2.745	626187	263591	49.00	2.522
D17	630918	275871	46.36	2.495	646193	283134	55.59	3.758
D18	648849	353940	127.71	2.449	648947	354046	137.55	2.482
D19	652020	158999	44.48	3.470	654912	160728	48.57	2.648
D20	678206	201844	48.22	5.044	678258	202009	47.55	4.400
Geomean	295458	88899	80.59	1.057	296373	89416	82.77	1.078
Ratio	**1**	**1**	**1**	**1**	**1.010**	**1.020**	**1.097**	**1.008**

Table 6.2. Comparison of the new retiming with the incremental retiming in [12] implemented in ABC.

Design	**Profile**			**Proposed retiming**		**Retiming in** [12]	
	LUT	**FF**	**Lev**	**Lev**	**Time (sec)**	**Lev**	**Time (sec)**
N1	774721	201816	65	44	9.69	53	62.88
N2	591403	186258	92	36	35.89	36	236.08
N3	742411	259844	106	43	53.92	43	291.56
N4	789908	228312	65	42	10.79	47	78.19
Geomean			**80.12**	**41.13**	**21.21**	**44.31**	**135.63**
Ratio1			**1**	**0.513**		**0.553**	
Ratio2				**1**	**1**	**1.078**	**6.40**