A Practical Approach to Multiple-Class Retiming

Klaus Eckl
Institute of EDA
Technical Univ. of Munich
80290 Munich, Germany
klaus.eckl@ei.tum.de

Jean Christophe Madre
Synopsys, Inc.
8, Avenue de Vignate
38610 Gieres, France
madre@synopsys.com

Peter Zepter
Synopsys Inc.
700 E. Middlefield Rd.
Mountain View, CA-94043
zepter@synopsys.com

Christian Legl
Institute of EDA
Technical Univ. of Munich
80290 Munich, Germany
christian.legl@ei.tum.de

Abstract

Retiming is an optimization technique for synchronous circuits introduced by Leiserson and Saxe in 1983. Although powerful, retiming is not very widely used because it does not handle in a satisfying way circuits whose registers have load enable, synchronous and asynchronous set/clear inputs. We propose an extension of retiming whose basis is the characterization of registers into register classes. The new approach called multiple-class retiming handles circuits with an arbitrary number of register classes. We present results on a set of industrial FPGA designs showing the effectiveness and efficiency of multiple-class retiming.

1 Introduction

Retiming is a powerful optimization technique for synchronous circuits that was introduced by Leiserson and Saxe in 1983 [8]. It consists of moving the sequential elements (registers) in a circuit while preserving its I/O behavior. Retiming can be used (1) to reduce the clock period of a circuit (*minperiod* retiming) and (2) to reduce its number of registers while achieving a given clock period (*minarea* retiming); the latter is of most practical interest.

Since the seminal work by Leiserson and Saxe [8, 9], many researchers have contributed to the theoretical and practical aspects of retiming. Originally designed to handle edge-triggered flip-flops, retiming has been extended to also handle multi-phase level-clocked latches [6, 10]. Efficient implementations [16, 12, 11] have made retiming applicable to large circuits. Important contributions have been made to apply retiming to circuits with reset states [19, 4, 18, 13]. Finally, it has been shown that retiming can be used together with existing combinational optimization techniques [14, 2, 3, 15, 5], to further improve circuit performance.

Despite its proved effectiveness and efficiency, retiming has not been very widely used in industrial logic synthesis tools. One of the main technical reasons for this is that most available retiming packages do not handle in a satisfying way the circuits that engineers really design today. In practice, these packages work well on circuits whose registers do not have synchronous or asynchronous set/clear inputs, as well as no synchronous load enable input.

However, most modern technologies offer registers with asynchronous, and/or synchronous reset inputs, as well as a synchronous load enable input (also called clock enable). For instance, every logic block in a Xilinx XC4000 FPGA contains two D-type edge-triggered flip-flops with asynchronous reset and synchronous load enable inputs which can be connected to arbitrary signals [20]. As shown by the results presented in Section 6, fully exploiting these capabilities is absolutely mandatory to achieve high design quality.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 99, New Orleans, Louisiana
(c) 1999 ACM 1-58113-109-7/99/06..$5.00

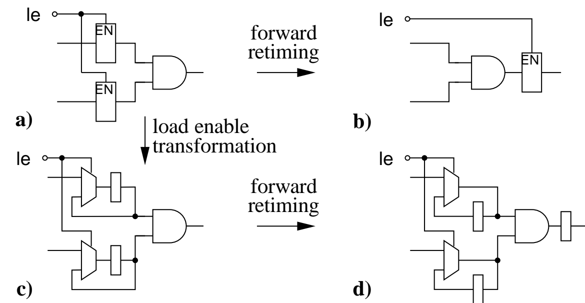

Figure 1: Two ways of retiming registers with load enables.

This is illustrated in Fig. 1 on a circuit that has two registers with load enable inputs. To apply existing retiming approaches, complex registers are transformed into simple registers with some additional logic to implement the synchronous load enable and reset behaviors. This transformation transforms the circuit a) into c), which is larger than a). Note that a register with asynchronous reset input has no equivalent synchronous circuit with a simple register and additional logic. Moving the simple registers forward results in circuit d). It can be seen that applying this retiming step results in an additional area cost of two registers and two multiplexors.

Camposano and Plöger showed [1] that registers can be moved *together* with their load enable inputs if they are connected to the same load enable signal. For instance, both registers in Fig. 1a) have the same synchronous load enable signal and thus can be moved forward together with their EN input to produce circuit b) which is much smaller than circuit d). Similar conditions for registers with asynchronous and synchronous reset inputs were presented by Singhal et al. [18]. However, both works only discuss the conditions for a single retiming step and do not present a comprehensive approach for computing a retiming solution. A first general approach to this problem was proposed by Legl et al. in [7], but they did not present any implementation showing that *minperiod* and *minarea* could both be solved in an effective and efficient way.

In this paper we present a practical and comprehensive approach called *multiple-class retiming*, or *mc-retiming*, which allows to efficiently and effectively compute a minperiod or minarea retiming solution for circuits designed with a variety of different registers. MC-Retiming is an extension of retiming that manipulates complex registers. The registers are classified into *register classes* which are used to determine how far backward and forward each register can be moved in the circuit. This information is then used to map the problem of multiple-class retiming into a *basic* retiming problem which can be efficiently solved using existing retiming approaches. Thus, the big advantage of mc-retiming is that it can reuse many of the efficient techniques available for basic retiming.

After giving background information on basic retiming in Section 2, we introduce in Section 3 the multiple-class retiming problem using a retiming graph model in which we classify the registers into register classes. In Section 4 we show how to map the multiple-class retiming problem into a basic retiming problem. Sec-

tion 5 presents an efficient implementation of multiple-class retiming in which we reuse existing basic retiming approaches. Finally, we present in Section 6 experimental results obtained with this implementation on a set of industrial FPGA designs.

2 Basic Retiming

The basic retiming approach presented by Leiserson and Saxe [9] handles sequential circuits whose registers are controlled by a single clock and possibly have reset values. A sequential circuit is represented by a vertex-weighted, edge-weighted, directed graph $G = (V,E,d,w)$, called *retiming graph*. Each combinational gate and each primary input and output port is modeled by a vertex $v \in V$. An edge e_{uv} models a connection from an output of gate u to an input of gate v, passing through an arbitrary number of registers. A *host vertex* v_h is introduced to model the environment of the circuit, together with edges from v_h to all primary inputs, and edges from all primary outputs to v_h. The vertex weight $d(v)$ is the propagation delay of the corresponding gate v. The edge weight $w(e_{uv})$ is the number of registers along the interconnection from gate u to gate v. For a path $p : u \rightsquigarrow v$ from vertex u to vertex v, the path weight $w(p)$ is the sum of the edge weights along the path, and the path delay $d(p)$ is the sum of the vertex delays along the path including the delays of vertices u and v. Finally, $W(u,v)$ denotes the minimum path weight of all paths from u to v, and $D(u,v)$ denotes the maximum path delay among all minimum weighted paths from u to v.

A *retiming* of G is an integer-valued vertex labeling $r : V \rightarrow Z$. By definition, the retiming (or lag) value $r(v)$ is the number of layers of registers moved from the fanout to the fanin edges of v. If $r(v)$ is negative, registers are actually moved from the fanin to the fanout edges of v. The edge weights after applying r to G are calculated using

$$w_r(e_{uv}) = w(e_{uv}) + r(v) - r(u).$$

A retiming is *legal* if all edge weights $w_r(e_{uv})$ are nonnegative (*circuit constraints*). A clock period ϕ is said to be *feasible* for G if there exists a legal retiming of G such that any path with $D(u,v) > \phi$ has at least one register on it in the retimed graph (*period constraints*). A clock period is thus feasible if there exists a retiming that satisfies the linear difference constraints

$$\begin{array}{llll} \text{circuit constr.:} & r(u)-r(v) & \leq & w(e_{uv}), \quad \forall e_{uv} \in E \\ \text{period constr.:} & r(u)-r(v) & \leq & W(u,v)-1, \quad \forall D(u,v) > \phi. \end{array}$$

The satisfiability of these constraints and an appropriate set of retiming values can be efficiently computed, e.g., by the *FEAS* algorithm [9]. Using the FEAS algorithm and binary search, it is easy to compute the minimum feasible clock period ϕ_{min}.

To solve the minimum area retiming problem, Leiserson and Saxe introduced a cost function that takes into account the possible sharing of the registers on the different fanout edges of each vertex [9]. This cost function, together with the circuit and period constraints, forms a special integer linear program (ILP) whose solution can be computed using a minimum-cost flow algorithm [9]. Recently, very efficient reduction techniques have been presented for this ILP formulation resulting in a significant speedup [16, 12, 11].

3 Multiple-Class Retiming

This section shows how a circuit with complex registers can be retimed without transforming these registers into simple registers and additional logic. It introduces register classes and explains how classes are added to a retiming graph.

3.1 Retiming Circuits with Multiple-Class Registers

Most sequential elements in synchronous circuits can be represented by the generic register shown in Fig. 2a). Each register has a signal connected to the data input D, the data output Q, and to the clock input. Additionally, a register can have inputs SS or SC and AS or AC which allow to synchronously and asynchronously set or clear

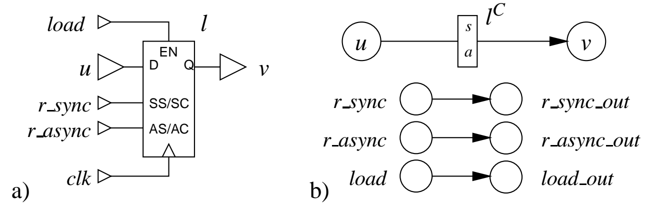

Figure 2: a) generic register l and b) corresponding mc-retiming graph with register l^C of register class C

the register, and a synchronous load enable input EN. If a register has, e.g., no load enable capability, then the synchronous load enable input EN of the generic register is deactivated by connecting it to a signal representing the constant 1.

Generic registers must fulfill certain conditions to be moved across a combinational logic gate. In general, such a retiming step is valid if it yields a circuit which is a *sufficiently old replacement* [8] of the original circuit. It has been shown in [1] that, for registers with synchronous load enable inputs, moving a layer of registers across a gate is valid if all registers are connected to the same load enable signal. The same condition holds for the clock inputs of the registers, because it is necessary to preserve the temporal equivalence of the circuit [17]. Registers with reset inputs can be moved if the reset signals are equivalent [18].

Since the validity of moving registers depends on the connected control signals, we classify the registers of a circuit using the signals connected to the control inputs.

Definition 1 (Register Class) *A register class C is characterized by a tuple (clk, load, r_sync, r_async) of signals. A register l belongs to class C iff each signal connected to its control inputs is logically equivalent to the corresponding signal of the class. Two registers are said to be* compatible *iff they belong to the same register class.*

It follows from this definition that a layer of registers can be moved across a logic gate if all registers are compatible.

3.2 Retiming Graph for Multiple-Class Circuits

A circuit which contains multiple register classes is called a *multiple-class circuit*. Since the validity of moving registers in a multiple-class circuit directly depends on which classes these registers belong to, we have to model the class information in the retiming graph. Especially, it is no longer sufficient to store only the number of registers $w(e)$ on an edge e of the retiming graph, as the registers on the edge may belong to different classes. Therefore, we introduce a modified retiming graph $G^{mc} = (V,E,d,\underline{l})$ which we call a *multiple-class retiming graph* or, in short, a *mc-graph*. Fig. 2b) shows how a generic register is modeled in the mc-graph. Instead of a weight $w(e)$, we attach to e a sequence of registers $\underline{l}(e) = [l_1, \cdots, l_{w(e)}]$. l_1 corresponds to the register closest to the source of the edge, while $l_{w(e)}$ is the register closest to the sink of the edge. The superscript C at a register l^C denotes the class to which it belongs. In the presence of reset inputs, a register is labeled with appropriate values $s, a \in \{0,1,-\}$ which specify the synchronous and asynchronous reset values of the register, respectively. For each control signal, except the clock signals, we introduce an output vertex in the mc-retiming graph and an edge from the vertex generating the signal to the corresponding output vertex. This is necessary to ensure that these signals get correctly handled through retiming.

A *valid mc-retiming step* for a vertex v can be performed as depicted in Fig. 3. For instance, for a forward mc-retiming step at vertex v, there must be a complete *layer* of *compatible* registers at the sink of the fanin edges of v. The last registers of the fanin edges are removed, and a new layer of registers with the same register class is inserted at the source of the fanout edges of v.

As in the basic retiming approach, we define a retiming for a mc-graph as an integer-valued vertex labeling $r : V \rightarrow Z$. A mc-retiming r is legal for a multiple-class circuit, if it can be implemented by a sequence of valid mc-retiming steps.

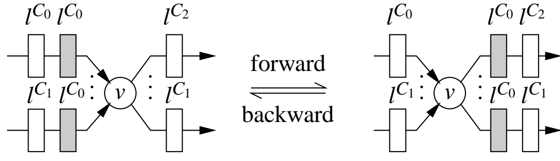

Figure 3: A valid multiple-class retiming step

4 Mapping Multiple-Class to Basic Retiming

This section presents the simple mechanisms that allow us to map the problem of retiming a multiple-class circuit onto the basic retiming problem which can then be solved efficiently by existing approaches to basic retiming.

4.1 Multiple-Class Retiming Constraints

A legal mc-retiming can only move layers of compatible registers. As a consequence, a value $r(v) > 0$ is valid for the vertex v of a mc-graph if there exist $r(v)$ complete layers of registers on the fanout paths of vertex v, and if each layer is made of compatible registers. This ensures that we can perform $r(v)$ valid mc-retiming steps backward across vertex v. This also means that the number of complete layers of compatible registers in the transitive fanout of v determines the maximum valid retiming value which we denote *backward mc-retiming bound* $r^{mc}_{max}(v)$. By analogy, a value $r(v) < 0$ is valid if there exist $|r(v)|$ complete layers of compatible registers on the fanin paths of vertex v. If k is the number of complete layers of compatible registers in the transitive fanin of v, then the minimum valid retiming value is given by the *forward mc-retiming bound* $r^{mc}_{min}(v) = -k$.

These mc-retiming bounds can be used to express the conditions for a mc-retiming r to be legal:

$$\text{circuit constr.:} \quad r(u) - r(v) \le w(e_{uv}), \qquad \forall e_{uv} \in E \qquad (1)$$
$$\text{class constr.:} \quad r^{mc}_{min}(v) \le r(v) \le r^{mc}_{max}(v), \quad \forall v \in V. \qquad (2)$$

As in basic retiming, the circuit constraints ensure that retiming does not create negative edge weights. In addition, the class constraints guarantee that at each vertex v only valid mc-retiming steps are performed. Thus, we can consider a legal mc-retiming to be a legal basic retiming with additional constraints set on the retiming values.

The mc-retiming bounds can be easily computed on the mc-graph. Instead of traversing the register layers reachable in the transitive fanin or fanout of a vertex, we adopt a different procedure which was proposed in [7]. In order to compute the backward mc-retiming bounds, we move registers backward as long as we can apply valid mc-retiming steps in the graph. Thereby, we count the number of registers which are moved across each vertex. When no more valid backward moves are possible, the mc-retiming graph is *maximally* backward retimed, and the number of registers moved across each vertex v is equal to the backward mc-retiming bound $r^{mc}_{max}(v)$. Similarly, to compute the forward mc-retiming bounds, we move the registers forward as far as possible using valid mc-retiming steps only. In the maximally forward retimed graph the negative number of registers moved across a vertex v equals the forward mc-retiming bound $r^{mc}_{min}(v)$.

Note that we do not consider reset values while computing the retiming bounds. Although this may result in maximal backward retiming bounds which can actually not be achieved due to justification conflicts, we decided to ignore reset values for two reasons. First, it was shown in [13] that retiming constraints which guarantee justifiable reset values are generally not unique resulting in a large number of different constraint sets. Thus, in order to find the optimal solution a retiming must be computed for each constraint set. Second, the backward justification of reset values can computationally be very expensive. Thus, we want to justify only those backward retiming steps which are actually required by the retiming solution. Our experiments have shown that the number of required backward retiming steps is usually much smaller than the number of retiming steps performed during maximal backward retiming.

Thus, by not considering reset states we compute a unique set of class constraints. Only when implementing the retiming solution do we compute equivalent reset states and take appropriate action in case of a justification conflict. Section 5.2 gives more details on how we compute equivalent reset states.

4.2 Register Sharing for Multiple-Class Registers

Minimum area retiming requires that we take register sharing at the gate output into account to get correct area estimation. The problem here is that if we directly apply the cost function introduced by Leiserson and Saxe [9] to count registers in the mc-graph, this would produce a register count that would be smaller than the actual count. Indeed, registers belonging to different classes cannot be shared. In the example in Fig. 4a) we would report a shared register count of 2. But the registers of class C_1 and C_2 cannot be shared so that the area cost is actually 3.

Recently, Maheshwari and Sapatnekar proposed an extended register sharing model [13] which takes into account restricted sharing due to different reset values. Their model could also be adapted to work with multiple-class retiming. However, this model results in a general 0/1-MILP retiming formulation which is much more expensive to solve than a minimum-cost flow problem. We suggest a new approach in which the graph is modified so that the register count is no longer underestimated by the sharing cost function of Leiserson. The resulting problem can still be solved using an efficient minimum-cost flow algorithm.

In a mc-graph, the sharing cost function underestimates the register count if registers of different classes appear in a register layer on the fanout edges of a vertex. In Fig. 4a) the second register layer gives an example for this case. In order to detect these cases, we make the following two observations. First, any register layer which results from a forward move across a multiple-fanout vertex can be unrestrictedly shared at the fanout edges because all inserted registers belong to the same class. Second, any register layer which can be moved backward across a multiple-fanout vertex can also be shared. Otherwise, it could not be moved backward. Thus, the shared register count is potentially wrong only for those registers which are in their maximal backward position.

Fig. 4b) shows the example mc-graph with its registers in the maximal backward position. The backward mc-retiming bounds are depicted at the vertices. In order to estimate the shared register count, we heuristically identify the largest number of sharable registers and separate them from the remaining registers. The set of sharable registers is found by traversing the register layers from the sources to the sinks of the fanout edges. At each layer, we select the registers that constitute the largest set of compatible registers. Then, we proceed to the next layer using only the edges of the recently selected registers. In Fig. 4b), all registers on the left side of the cutline can be shared while the registers on the right side of the cutline cannot be shared with any register on the left side.

Our goal is to forbid the registers that are at the right of the cutline to move onto the fanout edges of u where they would be considered as sharable by the area cost function. To do this, we introduce a separation vertex s_i with zero delay on each edge e_{uv_i} along the cutline. Thereafter, each non-sharable register is placed on the edge of a single-fanout vertex and is thus counted as one register. We prevent the non-sharable register to move backward across the separation vertices by specifying appropriate backward retiming bounds. If $w_b(e_{s_iv_i})$ denotes the weight of the edge $e_{s_iv_i}$ after maximal backward retiming, then the backward retiming bounds of a vertex s_i is given by

$$r^{mc}_{max}(s_i) = \max(r^{mc}_{max}(v_i) - w_b(e_{s_iv_i}), 0). \qquad (3)$$

Informally, if we rewind the maximal backward retimed graph to its starting position, then $r^{mc}_{max}(s_i)$ is the number of registers that have to pass the cutline in order to undo the maximal backward retiming at vertex v_i. Using this procedure, we also find how the initial registers must be distributed on the edges e_{us_i} and $e_{s_iv_i}$. Fig. 4c) shows how the initial mc-graph is finally modified to account for multiple-class

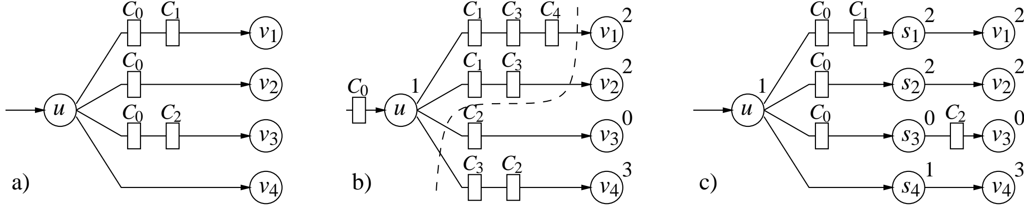

Figure 4: Modification for register sharing; a) initial mc-graph, b) maximal backward retiming; the cut separates a maximal set of sharable registers, c) mc-graph modified for register sharing.

register sharing. Note that each register which enters an edge $e_{s_iv_i}$ from vertex v_i during retiming is immediately passed to edge e_{us_i} as long as $r(s_i) < r^{mc}_{max}(s_i)$. This is because a register placed on e_{us_i} has a lower cost than a register placed on $e_{s_iv_i}$.

The above transformation is performed at each multiple-fanout vertex before solving the minarea retiming problem. It must be noted that there are certain situations where the register count is overestimated by our approach. If, e.g., in Fig. 4b) the registers on the edge e_{uv_4} swap their classes, then the first registers of e_{uv_3} and e_{uv_4} could be shared. This is not detected by our sharing model, because it separates only the largest set of sharable registers at a multiple-fanout vertex. However, these cases occur only if registers are in maximal backward position which does not seem to happen very often in practice. Furthermore, it is more desirable to overestimate the area during retiming rather than to underestimate it.

5 Efficient Implementation

A technically relevant implementation of multiple-class retiming must be able to compute a minimum area retiming for a minimum feasible clock period. This is achieved by performing the following steps which summarize the overall mc-retiming approach:

1. Generate the mc-graph G^{mc} from the circuit description.
2. Derive the retiming bounds $r^{mc}_{max}(v)$ and $r^{mc}_{min}(v)$ using maximal backward and forward retiming, respectively.
3. Modify the retiming graph so as to improve the estimation of the shared register count during minarea retiming.
4. Compute a minimum period retiming subject to the retiming bounds to get the minimum feasible clock period ϕ_{min}.
5. Compute a minimum area retiming subject to the minimum feasible clock period ϕ_{min}.
6. Relocate the registers in the circuit according to the computed retiming values. Thereby, compute an equivalent synchronous and asynchronous reset state.

We have already discussed Steps 1 – 3 in the previous sections. These steps are performed very fast, especially since we do not consider reset states during maximal backward retiming. In the remainder of this section we focus on how to efficiently compute the retiming solutions and the equivalent reset states for the retimed multiple-class circuit.

5.1 Computing a Multiple-Class Retiming Solution

The previous sections show that we can view the mc-retiming problem on the mc-graph G^{mc} as a basic retiming problem where upper and lower bounds are imposed on the retiming values. Additionally, the graph G^{mc} is modified by introducing separation vertices to provide a more reasonable estimation of the shared multiple-class register count. Thus, the mc-retiming problem can be solved by any retiming approach as long as the retiming bounds are satisfied.

We implemented basic *minperiod* and *minarea* retiming using the efficient algorithms presented by Shenoy and Rudell [16]. These algorithms, however, cannot directly handle retiming bounds set on vertices. To overcome this limitation, we rewrite the corresponding class constraints in (2) as a set of difference constraints using the retiming value of the host vertex. We can assume $r(v_h)$ to be 0, since registers are not allowed to move across inputs and outputs of the circuit. Thus, from (2) we get two difference constraints which are $r(v_h) - r(v) \leq -r^{mc}_{min}(v)$ and $r(v) - r(v_h) \leq r^{mc}_{max}(v)$. These difference constraints can be modeled by an edge from the host vertex v_h to v with weight $w(e_{v_hv}) = -r^{mc}_{min}(v)$ and an edge from v to v_h with weight $w(e_{vv_h}) = r^{mc}_{max}(v)$. The complete retiming problem to be solved during minarea retiming for a target clock period ϕ in Step 5 is then given by the following ILP formulation:

$$\min \sum_{v \in V} c(v) \cdot r(v)$$

$$\begin{array}{rrcll} \text{subject to} & & & & \\ \text{circuit constr.:} & r(u) - r(v) & \leq & w(e_{uv}), & \forall e_{uv} \in E \\ \text{class constr.:} & r(v_h) - r(v) & \leq & -r^{mc}_{min}(v), & \forall v \in V \\ & r(v) - r(v_h) & \leq & r^{mc}_{max}(v), & \forall v \in V \\ \text{period constr.:} & r(u) - r(v) & \leq & W(u,v) - 1, & \forall D(u,v) > \phi. \end{array}$$

The cost coefficient $c(v)$ is determined for each vertex v according to the sharing cost model of Leiserson and Saxe [9]. In order to solve the minimum period retiming problem of Step 4, the cost function is omitted and the minimum clock period ϕ_{min} resulting in a feasible set of constraints is determined by binary search.

Note that the number of class constraints is small compared to the possibly huge set of period constraints. The algorithm presented in [16] already makes use of efficient techniques to reduce the number of period constraints of which many are redundant. We expect to further reduce the overall number of constraints by using the technique proposed by Maheshwari and Sapatnekar [12, 11]. They showed that additional bounds on retiming values can be effectively used to further prune the set of constraints resulting in a much smaller ILP.

5.2 Computing Equivalent Reset States

Our technique for reset state computation is similar to the one proposed by Even et al. [4]. They move registers across several logic gates and then compute new reset values using forward implication or backward justification across the retimed logic gates. These steps are iterated until all registers are in their final position.

Since backward justification can be very expensive, our idea is to break down the justification task into justification steps as easy to execute as possible, as long as this provides a valid solution. Only if this simple approach fails to find a justification, do we perform a possibly more expensive justification. This mechanism is the following. Like [4], we concurrently compute a new reset state while moving registers into their final position. However, we compute new reset values each time a layer of registers is moved across a gate, which means that we just have to justify across one gate at a time, which is usually not expensive. This operation has been implemented using BDDs.

In a backward justification step we select as many don't cares for the reset values as possible. This helps to avoid conflicts in subsequent backward justification steps and also improves the register sharing potential. If a justification conflict occurs, we try to resolve the conflict by a *global justification* step. In this case, we trace the conflicting registers back to their original positions together with other registers involved in moving backward the conflicting registers. Then, we try to compute a justification for the larger portion of logic gates. On success, we update the reset values and proceed.

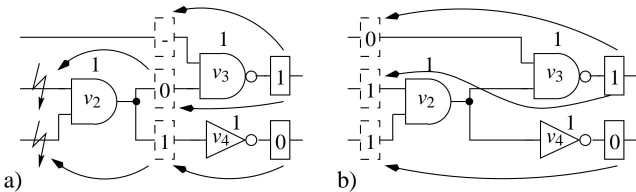

Figure 5: Computing reset values a) by local justification, b) by global justification.

If we cannot resolve the conflict by global justification, the retiming solution cannot be implemented, and we have to compute a new retiming solution. Beforehand we set an upper retiming bound on the vertex where the conflict occurred such that the non justifiable backward move is no longer allowed.

Fig. 5 illustrates our approach with an example. The numbers above the gates denote the retiming values to be applied to the circuit. The first two steps consist of moving the registers across the NAND gate v_3 and the inverter v_4, and local justification produces reset values for the registers inserted on the fanin edges of v_3 and v_4 (see Fig. 5a)). The following backward move across the AND gate v_2 produces a conflict due to the different reset values on the fanout edges. Therefore, the registers are traced back to the original registers and a global justification is performed across gates v_2, v_3, and v_4, as depicted in Fig. 5b).

In the experiments, our approach has shown to be very efficient. In less than only 1% of all justification steps we had to resort to global justification step in order to resolve a conflict. More impressively, we never encountered an example where we actually had to compute a new retiming solution due to a non-resolvable conflict. This shows that in practice computing equivalent reset states can be done in reasonable time using rather simple methods.

6 Results

We have developed a software package implementing the multiple-class retiming that we have presented here. As mentioned in Section 5, this package has been built on top of the efficient basic minimal period and minimal area retiming engine presented in [16]. Backward justification has been implemented using BDDs. This section presents the experimental setup that we have used to evaluate this new package, and then gives the results that we have obtained on real life industrial circuits using this setup. Note that it would not make sense to give results on standard benchmark circuits, like the ISCAS circuits, because they do not contain complex register and are also not available as RT-level HDL source, from which we could derive complex registers using an HDL analyzer.

The multiple-class retiming package has been integrated in an existing state-of-the-art logic synthesis system for FPGAs. This system provides us with scripts to perform logic synthesis, optimization and mapping of circuits for minimal area as well as for minimal area for best delay. Both the logic optimization and mapping are architecture specific, i.e., they both make use of the specific features of the target FPGA architecture to produce higher quality results. For instance, when mapping logic and arithmetic operators on a Xilinx XC4000E [20], it makes use of the hardwired carry chain logic to get the best performance.

Each circuit used here is an industrial circuit described at the RT-level in VHDL or in Verilog. This source code is first run through an HDL analyzer which produces a technology independent gate level netlist. Remarkable elements of this netlist are the registers, which can have a synchronous load enable input EN, as well as synchronous SS/SC and asynchronous AS/AC set/clear inputs.

Table 1 gives the areas and delays of each circuit after optimization, mapping, place and route, onto a Xilinx XC4000E, using the minimal area for best delay script. Since registers on a XC4000E do not have synchronous set/clear inputs, all such inputs inferred by the HDL analyzer are decomposed into additional logic before the optimization and mapping. On the other hand, all asynchronous set/clear and synchronous load enable inputs are used during mapping. For each circuit, columns **AS/AC** and **EN** indicate whether the circuit contains registers with asynchronous set/clear and synchronous load enable inputs respectively. Column **#FF** is the number of registers in the circuit. Column **#LUT** is the number of lookup tables (LUT) in the mapped circuit, and **Delay** is the minimal period of the circuit. This delay is the maximal delay over all combinational paths in the circuit, computed **after** place and route using Xilinx timing analyzer [20].

Table 1: Circuit Characteristics

Name	AS/AC	EN	#FF	#LUT	Delay
$C1$	√	√	35	89	32.4
$C2$	√	√	12	69	28.2
$C3$		√	26	48	27.4
$C4$		√	301	1185	160.4
$C5$	√	√	88	140	29.1
$C6$	√		1027	1268	72.4
$C7$	√	√	315	523	37.6
$C8$		√	79	145	36.1
$C9$	√	√	79	416	73.3
$C10$	√	√	206	885	48.1
Totals			2168	4768	545.0

Table 2: Retiming Results

Name	#Class	#Step	#FF	#LUT	Delay	Rlut	Rdelay
$C1$	8	6/106	48	115	29.1	1.29	0.90
$C2$	3	14/63	23	70	24.5	1.01	0.87
$C3$	4	3/35	29	66	24.4	1.38	0.89
$C4$	11	285/1528	342	1135	97.9	0.96	0.61
$C5$	15	11/149	88	134	26.7	0.96	0.92
$C6$	1	7/1889	1080	1268	51.6	1.00	0.71
$C7$	40	56/728	337	534	36.3	1.02	0.97
$C8$	7	2/97	102	140	34.6	0.97	0.96
$C9$	6	15/1017	111	467	62.7	1.12	0.86
$C10$	5	31/1508	219	710	39.7	0.80	0.83
Total			2379	4639	427.5	0.97	0.78

In order to evaluate the new retiming package, the mapping script was modified to include a retiming step. A command "retime" was inserted after the circuit has been completely mapped. The circuit is then seen as a netlist of Xilinx primitives, e.g., LUTs, carry chains, and special buffers. We decided to run retiming at this point because it allows us to compute delays for the combinational gates that are as close as possible to the actual delays in the FPGA. This is particularly important when dealing with carry chains for instance. The command "retime" is run with the minimal area for best delay objective. A command "remap" was also added to the script to remap the combinational part of the circuit after retiming.

Table 2 presents the results obtained with this modified script. The first part provides information about the retiming process itself, while the second part provides information about its effect. Column **#Class** is the number of classes in the mc-graph of the circuit. In column **#Step** the first number is the total number of layers of registers that have been actually moved in the circuit. The second number is the total number of all possible valid mc-retiming steps in the mc-graph, computed during the maximal backward and forward retiming phase. Column **#FF** is the number of registers in the retimed circuit, **#LUT** its number of LUTs, and **Delay** its maximal combinational delay. Finally, **Rlut** and **Rdelay** are the ratio of columns **#LUT** and **Delay**, respectively, over the corresponding columns of Table 1.

First of all, the overall "retime" command finished for all circuits within 60 seconds of CPU time on a Sun Ultrasparc (333 Mhz), showing the efficiency of our retiming approach. On average, about 90% of the time was used by the basic retiming approach, and 7% of the time was spent in register relocation and reset state computation. Only 3% of CPU time was used for building the mc-graph, computing the classes and retiming bounds, and modifying the graph for register sharing. This shows that the computational overhead caused

Table 3: Retiming Results without using Load Enable Inputs

Name	#FF	#LUT	Delay	$\mathbf{Rlut_1}$	$\mathbf{Rdelay_1}$	$\mathbf{Rlut_2}$	$\mathbf{Rdelay_2}$
*C*1	35	103	30.2	1.16	0.93	0.90	1.04
*C*2	23	71	24.8	1.03	0.88	1.01	1.01
*C*3	35	72	24.7	1.50	0.90	1.09	1.01
*C*4	453	1422	86.7	1.20	0.54	1.25	0.89
*C*5	94	173	30.1	1.24	1.03	1.29	1.13
*C*6	1080	1268	51.6	1.00	0.71	1.00	1.00
*C*7	363	748	37.3	1.43	0.99	1.40	1.03
*C*8	119	192	42.4	1.32	1.17	1.37	1.23
*C*9	122	477	63.3	1.15	0.86	1.02	1.01
*C*10	214	728	39.6	0.82	0.82	1.03	1.00
Totals	2538	5254	430.7	1.10	0.79	1.13	1.01

by our extension to retiming is very small.

Note that for all processed designs, the number of register layers actually moved is much smaller than the number of layers that can possibly be moved. Also note that over 99% of all needed backward justifications could be performed locally. This means that the cost of backward retiming is kept as low as possible. Although of course this cannot always be the case, we think that this is still very encouraging in practice.

Retiming proves to be fairly effective, with the largest delay reductions being obtained for the three largest circuits (C4, C6, C10). The penalty incurred on the combinational area by the process is non existent or very small for a majority of the designs, although 3 out of the 10 designs see their number of LUTs grow more than 10%. The penalty on the number of registers is more significant, with an average ratio of the penalty equal to 1.10.

In another experiment we compared the results presented in Table 2 with the results we obtain if we don't preserve the load enable inputs for retiming. In order to do so, we added at the beginning of the script a command that decomposes the synchronous load enable inputs of all the registers in the design. The results for this script are presented in Table 3. The first part of the table gives the number of registers, the number of LUTs, and the maximal delay in the circuit mapped using this modified script. These values are then compared with the values presented in Table 1 ($\mathbf{Rlut}_1$ and $\mathbf{Rdelay}_1$) and with the values presented in Table 2 ($\mathbf{Rlut}_2$ and $\mathbf{Rdelay}_2$).

The column $\mathbf{Rdelay}_2$ shows that there is only one circuit ($C4$) for which the resulting delay is better than the one given in Table 2. This can happen since after decomposing the load enable inputs there may be less restrictions in moving registers around resulting in a better delay improvement. For circuit $C4$ this comes, however, with a very significant area penalty of 32% more registers and 25% more LUTs. For all other designs, the delay in the retimed design is larger than the one reported in Table 2. Overall, after decomposing the load enable inputs retiming produces circuits that are 21% faster than the original circuits, but with 17% more registers and 10% more LUTs, while using the load enables during multiple-class retiming produces circuits that are 22% faster than the original circuits, with 10% more registers and 3% less LUTs.

7 Conclusion

In this paper we have presented an extension of the basic retiming algorithm which allows us to apply retiming on circuits designed to take advantage of the complex registers available in modern hardware technologies, such as registers with synchronous load enable, and synchronous and asynchronous set/clear inputs.

We have implemented this new retiming algorithm, called multiple-class retiming, integrated it in a state-of-the-art FPGA synthesis environment, and reported results obtained on a set of industrial FPGA designs. We think that these results are quite encouraging, because they show that the computational overhead caused by the extension is very small compared with its benefits on the processed circuits.

Acknowledgment

We are very grateful to P. Vanbekbergen, N. Shenoy, and A. Wang for many valuable discussions. K. Eckl and C. Legl also thank Synopsys Inc. for the opportunity of a summer internship and appreciate the continuous support and interest of Prof. K. J. Antreich in their work.

References

[1] R. Camposano and P. G. Plöger. Retiming and high-level synthesis. In *International Workshop on High-Level-Synthesis*, pages 191–201, Nov. 1992.

[2] G. De Micheli. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):63–73, Jan. 1991.

[3] S. Dey, M. Potkonjak, and S. G. Rothweiler. Performance optimization of sequential circuits by eliminating retiming bottlenecks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 504–509, Nov. 1992.

[4] G. Even, I. Y. Spillinger, and L. Stok. Retiming revisited and reversed. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(3):348–357, Mar. 1996.

[5] S. Hassoun and C. Ebeling. Experiments in the iterative application of resynthesis and retiming. In *ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Dec. 1997.

[6] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. In T. Knight and J. Savage, editors, *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 245–264. MIT Press, 1992.

[7] C. Legl, P. Vanbekbergen, and A. Wang. Retiming of edge-triggered circuits with multiple clocks and load enables. In *International Workshop on Logic Synthesis (IWLS)*, volume 1, May 1997.

[8] C. E. Leiserson and J. B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[9] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[10] B. Lockyear and C. Ebeling. Optimal retiming of multi-phase, level-clocked circuits. In T. Knight and J. Savage, editors, *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 265–280. MIT Press, 1992.

[11] N. Maheshwari and S. Sapatnekar. Efficient retiming of large circuits. *IEEE Transactions on VLSI Systems*, 6(1):74–83, Mar. 1998.

[12] N. Maheshwari and S. S. Sapatnekar. An improved algorithm for minimum-area retiming. In *ACM/IEEE Design Automation Conference (DAC)*, pages 2–7, June 1997.

[13] N. Maheshwari and S. S. Sapatnekar. Minimum area retiming with equivalent initial states. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 216–219, Nov. 1997.

[14] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(1):74–84, Jan. 1991.

[15] P. Pan and C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. In *ACM/IEEE Design Automation Conference (DAC)*, pages 720–725, June 1996.

[16] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 226–233, Nov. 1994.

[17] N. V. Shenoy, K. J. Singh, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. On the temporal equivalence of sequential circuits. In *ACM/IEEE Design Automation Conference (DAC)*, pages 405–409, June 1992.

[18] V. Singhal, S. Malik, and R. K. Brayton. The case for retiming with explicit reset circuitry. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 618–625, Nov. 1996.

[19] H. J. Touati and R. K. Brayton. Computing the initial states of retimed circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(1):157–162, Jan. 1993.

[20] Xilinx Inc., San Jose, California 95124. *The Programmable Logic Data Book*, 1996.