

Performance and Area Optimization using Sequential Flexibility

Christoph Albrecht¹

Pascal Witte^{2*}

Andreas Kuehlmann¹

¹ Cadence Berkeley Labs, Berkeley, CA, USA

² University of Hannover, Hannover, Germany

Abstract

Due to the ad-hoc specification methodology, typical ASIC designs are highly unbalanced with respect to the timing criticality of their logic paths. Traditional combinational synthesis does not support “borrowing” of timing slack across registers and therefore may result in a drastic overdesign of many paths and an overall loss of performance. This can be particularly harmful when paths are sped up through resizing which incurs an exponential cost in area. In this paper we present how clock latency scheduling interleaved with combinational optimization can be applied to optimize the performance and area of designs. In contrast to a retiming-based approach, we show that clock latency scheduling is computationally more efficient for inclusion in an inner synthesis loop and can also perform slack balancing which provides a good starting point for heuristic area minimization. Our preliminary experiments applying the presented flow based on an industrial synthesis tool to standard benchmark circuits and customer ASIC designs show an average improvement in performance of 17% and in area of 2%.

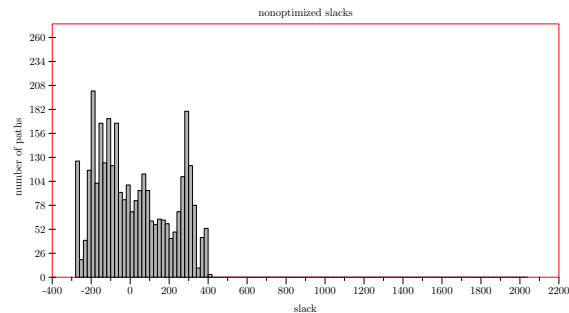
1 Introduction

Typical ASIC designs are highly unbalanced with respect to the timing criticality of their logic paths. This is mainly due to the ad-hoc manual design entry on register transfer level (RTL) which does not utilize any analysis feedback regarding the sequential criticality of design parts. Traditional combinational synthesis does not support “borrowing” of timing slack across registers and therefore may result in a drastic overdesign of many paths and loss of performance. This can be particularly harmful when paths are sped up through resizing which incurs an exponential cost in area.

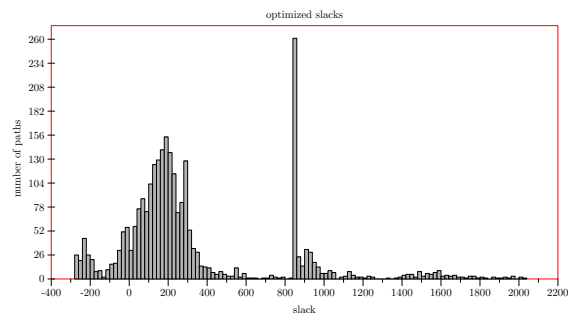
Sequential optimization techniques have been researched for many years and there are a number of efficient approaches available that are applicable to practical designs. Sequential synthesis methods of practical interest are retiming [1, 2] and clock latency scheduling [3]. In both cases, the goal is to balance the path delays between registers and thus to maximize the performance of the design without changing its input/output behavior.

Retiming is a structural transformation that moves the registers in a circuit without changing the positions of the combinational gates. Classical retiming does not address the problem of overdesigned logic paths because the commonly used formulations assume a fixed gate timing. For a given set of gate delays they either minimize the number of registers, maximize performance, or target some combination of the two objectives. None of them take into account the area sensitivity of logic paths with respect to delay. Retiming interleaved with synthesis [4] can distribute slacks in a limited manner. However, it still applies hard paths partitioning and thus may not predictably balance all slacks.

*This work was done while the author was with Cadence Berkeley Labs.



(a) Register latencies equal zero.



(b) Register latencies computed with the Slack Balancing Algorithm.

Figure 1: Combinational slack distributions for design s13207 (clock period 600ps), without (a) and with (b) register latencies.

In contrast to retiming, *clock latency scheduling* preserves the circuit structure, but applies tuned delays to the register clocks — thus virtually moving them in time. In recent years, clock latency scheduling has been adopted in a few design flows as a post-layout optimization technique to reduce the cycle time [5] and the number of close-to-critical paths [6, 7]. Clock latency scheduling can be viewed as a relaxed form of retiming which is computationally less complex. In fact, as shown in Section 5, an efficient implementation can determine an optimal schedule for large designs with tens of thousands of registers, making this form of sequential timing analysis well-suited for inclusion in the inner synthesis loop. Furthermore, the latency scheduling algorithm can simultaneously target optimal slack balancing which provides a good starting point for a heuristic area minimization approach.

In this paper we discuss the use of clock latency scheduling interleaved with synthesis as an alternative technique for optimizing the performance and area of a design. In Section 2 we motivate our work by providing slack statistics for a combinational and sequentially balanced view. The following section presents our algorithms for performance optimization. Section 4 describes how sequential flexibility is utilized for area optimization. Section 5 presents our computational results.

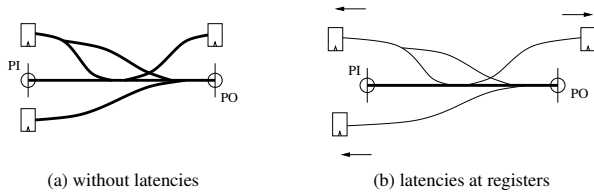


Figure 2: Example depicting a critical logic path from a primary input (PI) to a primary output (PO). Latencies at the registers can provide additional optimization possibilities to improve the delay of the PI-PO-path.

2 Motivation

As motivation for the utilization of sequential flexibility, this section discusses the slack distribution of a typical industrial ASIC design. Figure 1(a) depicts the slack distribution of the register-to-register paths in the circuit (paths with over 2200ps of slack were omitted). As shown, most paths are critical or near-critical in the combinational sense. Figure 1(b) gives the distribution of slacks for the same circuit, after clock latencies have been determined for all registers using the balanced slack algorithm described in Subsection 3.2. Note the strong contrast between the two histograms: the vast majority of the paths which were initially critical can be given significant flexibility using slack balancing through adjusting the register latencies.

The motivation of our work is to utilize this sequential flexibility for optimizing the performance and area during synthesis. For the shown example, without sequential flexibility, synthesis must assume that a large number of paths are close-to-critical, significantly restricting the optimizations scope. By utilizing sequential flexibility during optimization, the synthesis search space can be expanded, allowing substantial improvements in the circuit structure and overall gate sizing. This is best illustrated by noting that gate sizing algorithms dramatically increase the size of gates on the critical path for ensuring timing closure. For the given example, this would result in a significant number of oversized gates. A dedicated use of clock latencies can relax a large fraction of paths and thus achieve a substantial area savings.

Furthermore, the combination of clock latency scheduling and logic restructuring can enhance the circuit performance even in cases where none of the two methods alone can achieve any improvements. Figure 2 gives an example for this case. Suppose the given design has a critical path from a primary input to a primary output. Furthermore, assume that initially all paths from the primary input to all registers in its transitive fanout are critical and similarly all incoming paths to the output. This is indicated by the highlighted lines in Figure 2(a). Since all paths are critical, logic restructuring may not have any options to achieve timing closure beyond maximally sizing all gates.

However, with clock latency scheduling it may be possible to relax the timing requirements on the register-to-primary-output paths and primary-input-to-register paths by changing the latency of the registers as shown in part (b). The additional slack provides more options for logic restructuring that can focus on these paths only. For example, it may be possible to co-factor the logic function at the primary output with respect to the critical primary input [8].

This work does not consider the implementation of the different latencies at the registers. The latencies can be realized by retiming or by the construction of a sophisticated clock tree [7, 9].

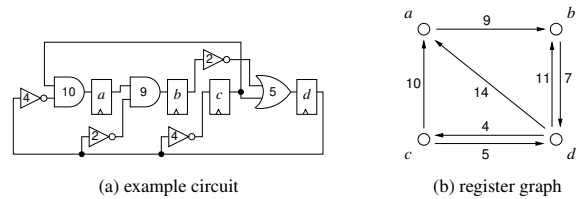


Figure 3: An example circuit and the corresponding register timing graph.

3 Performance Optimization

In this section we describe an iterative optimization approach for the *sequential performance optimization* problem:

Problem 1 Sequential Performance Optimization

Given: A sequential circuit.
Task: Synthesize the design and compute latencies $l(v)$ for all registers v .
Objective: Minimize the clock period T .

The algorithm iteratively extracts the maximum delay for each register-to-register path by combinational timing analysis, computes a clock schedule, adjusts the latencies at the registers, and then optimizes the design using the new synchronization latencies at the registers. Key to the performance and area improvements obtained is a clock schedule which balances the slack optimally. In the next two subsections we explain the algorithms and the special properties of the clock schedule computed. In the last subsection we present the overall optimization loop.

3.1 Critical Cycle Computation

The critical cycle is the cycle of registers and gates that determines the minimum achievable clock period (using retiming and/or clock latency scheduling techniques). In this subsection, we describe the process of finding the critical cycle. We illustrate the key concepts using the small example circuit shown in Figure 3(a).

Figure 3(b) shows the *register timing graph* corresponding to the circuit. Each node represents a register in the design. The graph has an edge between two nodes if there is a path between them. Associated with each edge is the maximum delay of all paths between the corresponding registers.

For a zero-latency clock schedule, the maximum clock period of the given circuit is 14 which is constrained by the delay between register d and a . Using clock latency scheduling we can adjust the register latencies to $a = 4$, $b = 3$, $c = 0$, and $d = 0$ (shown in Figure 4). With these new latencies, the example circuit can be clocked with a maximum clock period of 10. One can easily verify that, given this clock period, each edge in the graph still meets its timing requirements. For example, the edge from d to a still has a delay of 14, but since the clock at register a is delayed by 4 units, any signal along this edge will still meet the setup requirement at register a . The slack of each edge is given in brackets.

It turns out that for this example the given clock latencies are optimal, i.e., there is no clock schedule which allows the circuit to be clocked by a faster period. This is because the *critical cycle* of the circuit $a - b - d - a$ is optimally scheduled. Formally, the critical cycle is among all cycles of a circuit the one with the maximum mean delay, i.e., $total_delay/num_registers$.

For the following discussion, we formally define the register timing graph $G = (V, E)$ as follows. V represents all registers of the design and including additional vertex v_{ext} for all primary

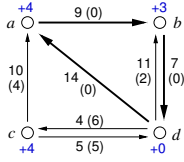


Figure 4: Example register timing graph with latencies applied. The critical cycle is highlighted.

inputs and outputs. The directed graph G has an edge $(u, v) \in E$ if and only if there exists a timing path from register u to register v in the circuit. Every edge $e = (u, v)$ is labeled with $d(e)$, the maximum delay between u and v in the circuit. We denote the clock period by T and the latency of a register v by $l(v)$.

Problem 2 Clock Latency Scheduling: Clock Period Minimization

Given: A register timing graph $G = (V, E)$, with a delay $d(e)$ for all $e \in E$.

Task: Find latencies $l(v)$ for all $v \in V$ and a clock period T such that

$$l(u) + d((u, v)) - T \leq l(v)$$

for all $(u, v) \in E$.

Objective: Minimize the clock period T .

Since the “latency” at a primary input or primary output cannot be changed, we need to find a solution with $l(v_{ext}) = 0$. However, this does not constrain our problem, since any solution can be transferred to a solution with $l(v_{ext}) = 0$ by adding a constant to all latencies $l(v)$, $v \in V$.

Finding the critical cycle is equivalent to computing the *maximum mean cycle* (MMC) for G , which is given by $\max_{C \in \mathcal{C}} \sum_{e \in C} d(e) / |C|$, where \mathcal{C} is the set of all cycles in G . The MMC is equal to the minimum clock period which may be obtained using clock latency scheduling. If we multiply the delay $d(e)$ of the edges by (-1) , then finding the maximum mean cycle in the original graph is equivalent to finding the minimum mean cycle in the modified graph. There are several algorithms to compute the minimum mean cycle in a directed graph. A good overview and an experimental comparison of the runtime of the different algorithms is given in [10]. We apply the algorithm by Young, Tarjan and Orlin [11] to compute the critical cycle and the clock schedule. This algorithm is used in [6, 12] and can be viewed as a modified version of Burn’s algorithm [13] which is applied in [9].

In the following we briefly outline the algorithm. For further details see [11, 6, 12]. The algorithm starts with a clock schedule which has a latency of zero for all registers. The clock period is set to the maximum delay of all edges in G . The algorithm decreases the clock period and simultaneously adjusts the latencies in order to keep a feasible clock schedule. It also maintains a set of critical edges which have zero slack.

In the beginning the set of critical edges consists only of the edges with the maximum delay. In order to decrease the clock period and to keep a feasible schedule the latencies of all registers which have a critical incoming edge are increased. In case there is a path of critical edges in G , for example $v_1, e_1, v_2, e_2, v_3, e_3, v_4$, the latency of v_1 is not changed, the latency of v_2 is increased at the same rate at which the clock period decreases, the latency of v_3 is increased at twice the rate of the latency of v_2 , similarly the latency of v_4 increases three times as fast. In general, if the

longest path of critical edges directed into vertex v consists of k edges, the latency $l(v)$ is increased at k times the rate at which the clock period decreases. As we know the rate at which the latency of each register changes, we can compute for any non-critical edge the clock period at which the edge becomes critical, assuming there is no other non-critical edge. With this information we determine the next edge that becomes critical. If a new edge becomes critical, the latency of the vertex into which the edge is directed has to start increasing at a faster rate. Subsequently, a previously critical edge directed into this vertex becomes uncritical. The algorithm maintains a set of directed path trees (arborescences) formed by the critical edges.

The algorithm stores for each vertex v the next incoming edge to become critical together with the corresponding clock period as key in a Fibonacci-Heap. The algorithm stops when the new critical edge forms a directed cycle with the critical edges. This cycle is the maximum mean cycle.

3.2 Balanced Slack

We have seen how a clock schedule minimizing the clock period can be computed. A clock schedule for the minimum clock period is not unique. Many registers may still have sequential flexibility. We would like to obtain a solution which optimally distributes the slack such that it is maximally balanced and logic synthesis has more optimization possibilities. In this subsection we will first formally define the Slack Balancing Problem and then present the algorithm, which extends the algorithm of the previous subsection.

The *slack* of an edge (u, v) is given by

$$slack((u, v)) = l(v) - l(u) - d((u, v)) + T.$$

The *incoming slack* of a register v is the minimum slack of all incoming edges, excluding possible loop edges (v, v) , and similarly, the *outgoing slack* of a register v is the minimum slack of all outgoing edges, excluding loop edges.

We will see that it is not sufficient to consider only the incoming and outgoing slack of single vertices. We extend this definition to subsets of vertices: For $V' \subseteq V$ we define the *incoming slack* of V' as

$$slack_{in}(V') = \min_{(u,v) \in E, u \in V \setminus V', v \in V'} slack((u, v)),$$

and similarly the *outgoing slack* of V' as

$$slack_{out}(V') = \min_{(v,u) \in E, v \in V', u \in V \setminus V'} slack((v, u)).$$

We define the Slack Balancing Problem as follows:

Problem 3 Clock Latency Scheduling: Slack Balancing

Given: A register timing graph $G = (V, E)$, with a delay $d(e)$ for all $e \in E$, and a clock period T .

Task: Find latencies $l(v)$ for all $v \in V$ such that

$$slack_{in}(V') = slack_{out}(V')$$

for all $V' \subset V$.

Objective: Minimize the clock period T .

The Slack Balancing Problem is equivalent to the minimum mean balance problem described in [11].

Figure 5 shows an example demonstrating that it is not sufficient to consider only the incoming and outgoing slack of single vertices: If the clock period is $T = 10$, the registers a and b must have the

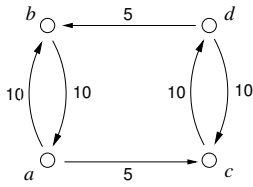


Figure 5: Register timing graph showing that it is necessary to consider subsets of vertices for the balance property.

same latency such that the edges (a, b) and (b, a) have zero slack. Similarly, the latencies of the registers c and d must be equal.

As long as the difference in the latencies of a and c is not greater than 5 the incoming slack and outgoing slack for each single vertex is zero. However, consider the incoming and outgoing slack of the subset $\{a, b\}$ or alternatively $\{c, d\}$: The condition of the Slack Balancing Problem requires that the slack of the edges (a, c) and (d, b) has to be equal. The only possible solution for the Slack Balancing Problem is that the slack of these edges is five and the latencies of all registers have to be equal.

We will present the algorithm to solve the Slack Balancing Problem:

Algorithm 1 Slack Balancing

- 1: **while** graph contains directed cycle **do**
- 2: find critical cycle C with maximum mean delay
- 3: assign latencies to C which satisfy timing
- 4: contract cycle C to a single vertex v_C
- 5: adjust delays on edges directed into and out of v_C
- 6: **uncontract** graph and compute latencies for all original vertices

There are two key ideas behind this algorithm. First, we make use of the fact that the vertices in a critical cycle always have their latencies in “lock-step” with each other. That is, such vertices have latencies which are all interdependent, and setting the latency for one vertex determines the latencies of all the others in the critical cycle. This is why in each iteration we *contract* the critical cycle C into a single vertex v_C , because we only need to compute a single latency for the representative (contracted) vertex to fully determine the latencies for the entire critical cycle. Second, the delays for edges directed into and out of v_C must be adjusted to account for the fact that the latencies assigned to the individual vertices in the critical cycle may be different from each other. That is, the incoming edges to the new vertex v_C may, in fact, have pointed to different vertices of C , and this must be reflected in the resulting graph after C is collapsed.

To illustrate this, we consider the same example register timing graph previously presented. The critical cycle is identified as (a, b, d) , and latencies of $(0, 4, 3)$ are assigned to these vertices as shown on the left hand side of Figure 6(a).

Vertices (a, b, d) are contracted to a single vertex abd , shown in the second part of Figure 6. We illustrate how the delays on the edges leading into and out of abd are adjusted as follows: consider the edge (c, a) . Let τ_v be the latency assigned to vertex v . Since a was assigned a latency of +4 before being contracted into abd , we can consider this vertex to have a latency of +4 relative to abd , so

$$\tau_a = 4 + \tau_{abd} \quad (1)$$

Now suppose the latency of c is set so that the edge (c, a) exactly meets its timing requirements. Then

$$\tau_c + 10 - \phi = \tau_a \quad (2)$$

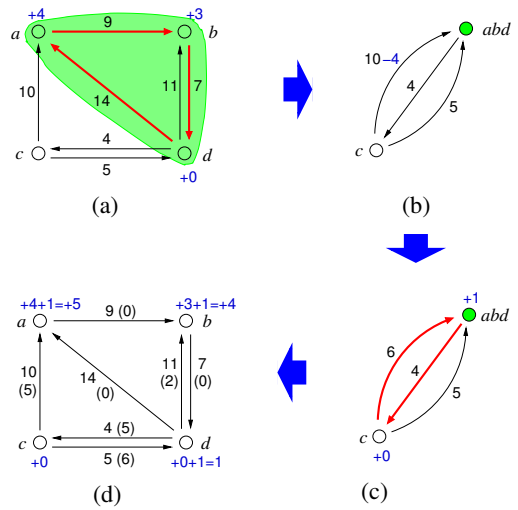


Figure 6: Intermediate steps of the Slack Balancing Algorithm.

where ϕ is the clock period. In addition, the edge (c, abd) must exactly meet its timing requirement, since abd represents a in the contracted graph. This results in:

$$\tau_c + q - \phi = \tau_{abd} \quad (3)$$

where q is the delay assigned to the edge (c, abd) . Combining Equations 1 through 3 we find $q = 6$.

The procedure of adjusting the edge delays can be easily generalized, such that edges directed into the contracted vertex have a value subtracted from their delays, where this value is the latency of the original vertex to which the edge was previously directed relative to the latency of the new vertex. Likewise, edges directed out of the contracted vertex have the corresponding latency added to their delays. This is illustrated in Figure 6(b).

After the cycle is contracted and the new delays are obtained, the process is repeated until there are no more cycles in the graph. Figure 6(c) shows the process repeated to obtain new latencies on the contracted graph.

The final step is to uncontract the graph and compute the latencies for the original vertices. This is illustrated in Figure 6(d). Since the latency of abd was assigned a value of +1 and a was given a latency of +4 relative to abd (in the first iteration), a must have an overall latency of +5. Similar computations are done for the other vertices. The slack of each edge of the final solution is shown in brackets.

As noted earlier, the final solution has the property that for each vertex the worst slack of all incoming edges equals the worst slack of all outgoing edges, hence we call the algorithm *balanced*. In the example, register c has a slack of +8, and can assume any latency between -6 and +4. The balanced slack algorithm assigns a latency of 0 to register c , balancing the slack such that half of this slack is distributed to the worst outgoing edge (c, a) and the other half to the worst incoming edge (d, c) .

For an efficient implementation the algorithm maintains the arborescences formed by the critical edges as the cycle with the current maximum mean delay is contracted. It does not have to start again from the beginning each time a cycle is contracted. It also efficiently updates the rates at which the latencies increase. For further details of the implementation see [11, 6, 12].

In practice, the algorithm does not need to run until all cycles are contracted, it can be stopped after a certain number of cycles

are contracted, or when a certain slack value is achieved. For the example, for which the slack distribution is shown in Figure 1(b), the algorithm stopped at a slack value of around 850 ps, and hence a large number of paths have this value.

3.3 Sequential Optimization Loop

We present the iterative optimization approach for the Sequential Performance Optimization Problem (Problem 1).

Algorithm 2 Sequential Performance Optimization

- 1: **repeat**
 - 2: analyze timing and extract register timing graph
 - 3: Slack Balancing (Algorithm 1)
 - 4: set synchronization latencies
 - 5: synthesize
 - 6: **until** terminated or no improvement
-

At the beginning the algorithm analyzes the timing and extracts the register timing graph. It then calls the Slack Balancing Algorithm followed by adjusting the register latencies. Then the design is optimized with “synthesize”. During “synthesize” we apply all optimization algorithms of a commercial synthesis tool. For our experiments in Section 5 we use the Cadence RTL Compiler.

In order to achieve performance improvements, we decrease the target period such that the slack becomes negative. In addition we use the option not to optimize only the worst slack, but as second criterion to optimize the incoming slack of the individual registers, also known as endpoint slack optimization. As third criterion the area is optimized subject to the constraint that all negative slacks do not decrease.

4 Area Optimization

In this section we consider the problem to minimize the area for a given clock period utilizing the flexibility to change the clock latencies at the registers.

Problem 4 Sequential Area Optimization

- Given:** A sequential circuit, a clock period T .
- Task:** Synthesize the circuit and compute latencies $l(v)$ for all registers v .
- Objective:** Minimize the area.
-

In the previous section, we use the Slack Balancing Algorithm for the Sequential Performance Optimization Problem to compute the latencies. It equally balances the incoming and outgoing slack for all subsets of registers. To reduce the area of the design, this is obviously not the best possible solution. For area minimization one needs to consider the area-delay sensitivity of the incoming and outgoing logic cone of each register and adjust the register latency such that both sensitivities are equal [14]. If the sensitivity is unbalanced as shown in Figure 7, one could simply decrease the area by increasing the delay for the side with a higher area/delay ratio at the expense of the delay of the other side.

We have developed a method, which iteratively changes the latencies of the registers and re-synthesizes the circuit. The method consists of three phases: *Initial Latency Shifts*, in which we change the latency of many registers simultaneously, then *Individual Latency Shifts*, which changes the latency of single registers and groups of registers and re-synthesizes the circuit only partially, and *Final Synthesize* in the end.

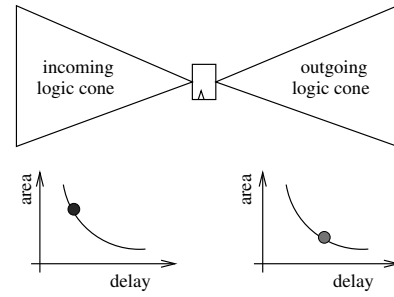


Figure 7: General concept of area minimization. The register latency must be adjusted such that the area-delay sensitivity of the incoming and outgoing cone is equal.

Algorithm 3 Sequential Area Optimization

- 1: Initial Latency Shifts
 - 2: Individual Latency Shifts
 - 3: Final Synthesize
-

We assume for the presentation of the algorithm that the slack values are not updated if the latency of a register is changed. However, the slacks are recomputed whenever the design is optimized by “synthesize”.

For a register v the direct successors in the register-to-register graph are denoted by $fanout(v)$ and the direct predecessors of v are denoted by $fanin(v)$.

Since the technology libraries have discrete gate sizes, it is possible that the incoming or outgoing slack of a vertex is positive, but any downsizing operation to decrease the area would result in a negative slack. However, if the slack is above a certain threshold δ , we assume that there is no area improvement possible. The combinational logic paths starting at or ending in this register are either timing constrained by other logic paths sharing some logic gates or the logic paths consist of gates with minimal size. We choose δ to be approximately half of the average gate delay of the given technology library. Decreasing the latency of a register which has incoming slack greater than δ and outgoing slack smaller than δ is likely to decrease the area.

Algorithm 4 Initial Latency Shifts

- 1: **for all** register v **do**
 - 2: **if** $slack_{in}(v) \geq \delta$ and $slack_{out}(v) < \delta$ **then**
 - 3: set $l(v) := l(v) - slack_{in}(v)$
 - 4: synthesize
 - 5: **for all** register v **do**
 - 6: **if** $slack_{in}(v) < \delta$ and $slack_{out}(v) \geq \delta$ **then**
 - 7: set $l(v) := l(v) + slack_{out}(v)$
 - 8: synthesize
-

Initial Latency Shifts are performed for all registers with slack above the threshold δ on one side and slack below this threshold on the other side. We call these registers *unbalanced registers*.

The algorithm moves many unbalanced registers simultaneously toward the point balancing the area-delay sensitivity. We perform the shifts in two steps to avoid causing a timing violation when shifting the slack from input to output or from output to input respectively. The first step shifts all available slack from the input of a register to the output. After synthesizing the design the remaining slack at the output of these registers is shifted back and the available slacks at the outputs are shifted to the inputs and the circuit is synthesized again.

Performing initial shifts can only cause the total area to shrink. The slack on the one side below δ is increased and the slack on the other side is decreased to zero. Decreasing the slack to zero just fits the allowed delay for logic synthesis to the actual delay of the logic paths.

After the initial latency shifts the latencies are shifted at the registers *individually*. For this, all existing logic is fixed and only the combinational logic surrounding the actual registers and the actual registers itself are allowed to be synthesized.

Function 5 `move_register(v, lnext, dir)`

```

1: set  $l_{orig} := l(v)$ 
2: repeat
3:   set  $l_{prev} := l(v)$ 
4:   set  $l(v) := l_{next}$ 
5:   synthesize;
6:   if ( $dir = "+"$ ) then
7:     set  $l_{next} := l_{next} + \Delta$ 
8:   if ( $dir = "-"$ ) then
9:     set  $l_{next} := l_{next} - \Delta$ 
10: until area increases or  $slack_{in}(v) + slack_{out}(v) < 0$ 
11: set  $l(v) := l_{prev}$ 
12: synthesize locally;
13: set  $l(v) := \max\{l(v), l(v) - slack_{in}(v)\}$ 
14: set  $l(v) := \min\{l(v), l(v) + slack_{out}(v)\}$ 
15: if ( $l(v) = l_{orig}$ ) then
16:   return false;
17: else
18:   return true;

```

Function 6 `process_register(v)`

```

1: if  $slack_{in}(v) \geq \delta$  and  $slack_{out}(v) < \delta$  then
2:    $moved := move\_register(v, l(v) - slack_{in}(v), "-")$ 
3: else if  $slack_{in}(v) < \delta$  and  $slack_{out}(v) \geq \delta$  then
4:    $moved := move\_register(v, l(v) + slack_{out}(v), "+")$ 
5: else
6:    $moved := move\_register(v, l(v) - \Delta, "-")$ 
7:   if ( $!moved$ ) then
8:      $moved := move\_register(v, l(v) + \Delta, "+")$ 
9: if ( $moved$ ) then
10:  for all register  $w \in fanin(v) \cup fanout(v)$  do
11:    set  $dont\_move(w) := false$ 
12: set  $dont\_move(v) := true$ 

```

During the *Individual Latency Shifts* shown in Algorithm 7 all unbalanced registers are shifted as single registers and registers with tight slacks at input and output are shifted in sets of registers. Registers with a $dont_move(v) = true$ and registers with a slack above δ on both sides are not considered for optimization.

Which register is handled next is determined by two criteria (Function 6). The first criterion is the worst slack in the design and the second is the difference between input and output slack. Therefore among the registers with the worst slack, the one with the greatest difference between input and output slack is optimized next.

For all registers with tight slacks on both sides, all registers constraining the same combinational logic are shifted in parallel. The set of registers that is shifted in parallel is determined as follows. We create two subsets of registers. The first subset consists of the *fanin* registers of the second level *fanout* combinational logic gates of the actual register. The second subset consists of the *fanout* registers of the second level *fanin* combinational logic gates of the

Algorithm 7 Individual Latency Shifts

```

1: for all register  $v$  do
2:   set  $dont\_move(v) := false$ 
3: repeat
4:   set  $V := \{v \mid dont\_move(v) = false \text{ and } (slack_{in}(v) \leq \delta \text{ or } slack_{out}(v) \leq \delta)\}$ 
5:   if  $V \neq \emptyset$  then
6:     Choose  $v \in V$  which minimizes  $\min\{slack_{in}(v), slack_{out}(v)\}$ , and as a second criterion maximizes  $|slack_{in}(v) - slack_{out}(v)|$ 
7:     process_register( $v$ )
8:   until  $V = \emptyset$ 

```

actual register. All registers that appear in both sets *and* that have tight slacks at input and output are shifted in parallel.

Function 6 shows in which directions the latency is shifted depending on the input and output slacks at a register and the threshold δ . The latency is shifted such that gaining slack at a register input or output may allow decreasing the respective logic paths. If the latency at a register changes, the *dont_move* flags at the fanin and fanout registers are set to *false*.

How the latency shifts itself are performed is shown in Function 5. The latency is shifted into one direction as long as the area decreases or the last synthesis step created an invalid result, i.e. $slack_{in}(v) + slack_{out}(v) < 0$ such that no clock latency can balance the slacks to a non-timing violating result.

The individual latency shifts are performed until the set of registers V that can be optimized is empty.

After the latencies have been optimized incrementally by considering individual registers, the entire design is synthesized once more. Changing the latency at a single register can influence the entire transitive fanout and transitive fanin, not only logic in the direct fanin and fanout. When searching for a latency only the direct fanin and fanout is re-synthesized. A final incremental synthesis of the netlist with the new latencies may therefore give further improvements.

5 Computational Results

In this section we describe the results obtained with the algorithms presented in this paper. We have implemented the Slack Balancing Algorithm (Algorithm 1) in C. For “synthesize” we use the Cadence RTL Compiler. It provides a Tcl-Interface which we used to extract the register timing graph and to set the latencies for the registers. The algorithm for area optimization presented in Section 4 is directly implemented in Tcl, since the algorithms frequently interact with their synthesis tool.

Table 1 shows the characteristics of our testcases. We use the five largest designs from the ISCAS benchmark suite (s15850, s13207, s38584, s38417, s35932) and 17 industrial customer designs (ind01, ..., ind17) for our experiments. The table gives the number of cells, the number of primary inputs (PIs), the number of primary outputs (POs), the number of registers (FFs), and the number of register-to-register paths.

In Table 2 we present the results obtained with Algorithm 2, Sequential Performance Optimization, which iteratively balances the slacks on the register-to-register paths and then optimizes the design with the Cadence RTL Compiler using the command “synthesize”. We show the target period in Column 3. The following columns give the minimum clock period which is equal to the target period minus the worst slack. In the fifth column we report

		1						2					
design	target period	start	synth.	clock	impr. (%)	synth.	impr. (%)	clock	impr. (%)	synth.	impr. (%)		
s15850	period area	750	872 30924	866 30824	864 30924	0.23 -0.32	767 29164	11.43 5.39	766 29164	11.55 5.39	762 28864	12.01 6.36	
s13207	period area	600	898 29017	893 28782	884 29017	1.01 -0.81	616 28798	31.02 -0.05	616 28798	31.02 -0.05	600 28675	32.81 0.37	
s38584	period area	780	905 73475	886 73668	894 73475	-0.90 0.26	788 72261	11.06 1.91	788 72261	11.06 1.91	785 71986	11.40 2.28	
s38417	period area	820	915 97783	899 97810	904 97783	-0.56 0.03	835 91038	7.12 6.92	835 91038	7.12 6.92	830 90836	7.68 7.13	
s35932	period area	400	437 130130	432 129639	437 130130	-1.16 -0.38	422 109730	2.31 15.36	422 109730	2.31 15.36	422 108764	2.31 16.10	
average	period area					-0.28 -0.25		12.59 5.90		12.61 5.90		13.24 6.45	
ind01	period area	7000	11448 20977	11447 20884	7782 20977	32.02 -0.45	7765 20687	32.17 0.94	7765 20687	32.17 0.94	7758 20683	32.23 0.96	
ind02	period area	1500	1982 35380	1982 35852	1901 35380	4.09 1.32	1768 35626	10.80 0.63	1719 35626	13.27 0.63	1681 35949	15.19 -0.27	
ind03	period area	2000	4869 139988	4868 139469	4674 139988	3.99 -0.37	3819 133881	21.55 4.01	3814 133881	21.65 4.01	3727 132580	23.44 4.94	
ind04	period area	1500	4828 119326	4824 119058	3167 119326	34.35 -0.23	2751 119032	42.97 0.02	2721 119032	43.59 0.02	2721 116829	43.59 1.87	
ind05	period area	2300	3354 233779	2930 212742	2759 233779	5.84 -9.89	2506 209846	14.47 1.36	2500 209846	14.68 1.36	2476 207186	15.49 2.61	
ind06	period area	4020	4351 192172	4340 175422	4310 192172	0.69 -9.55	4082 160023	5.94 8.78	4082 160023	5.94 8.78	4082 157564	5.94 10.18	
ind07	period area	2000	5299 211816	5285 210928	5286 211816	-0.02 -0.42	4977 211588	5.83 -0.31	4977 211588	5.83 -0.31	4910 211098	7.10 -0.08	
ind08	period area	3000	4208 1513197	4207 1512507	4207 1513197	0.00 -0.05	4181 1507750	0.62 0.31	4181 1507750	0.62 0.31	4192 1506649	0.36 0.39	
ind09	period area	1500	4496 486779	4496 486422	4486 486779	0.22 -0.07	3592 483158	20.11 0.67	3578 483158	20.42 0.67	3210 483464	28.60 0.61	
ind10	period area	3000	6011 2153	6011 2150	5984 2153	0.45 -0.14	5842 2137	2.81 0.57	5840 2137	2.84 0.57	5831 2135	2.99 0.69	
ind11	period area	1	909 832846	909 841176	836 832846	8.03 0.99	807 844753	11.22 -0.43	802 844753	11.77 -0.43	788 852799	13.31 -1.38	
ind12	period area	3000	4160 2632693	4160 2632693	4019 2632693	3.39 0.00	4019 2632693	3.39 0.00	4019 2632693	3.39 0.00	4019 2632693	3.39 0.00	
ind13	period area	6600	8479 1357512000	8382 1356588480	8420 1357512000	-0.45 -0.07	8168 1340539200	2.55 1.18	8168 1340539200	2.55 1.18	8106 1334472000	3.29 1.63	
ind14	period area	1000	6874 3274548	6874 3270773	5495 3274548	20.06 -0.12	5424 3223450	21.09 1.45	5162 3223450	24.91 1.45	5162 3187897	24.91 2.53	
ind15	period area	3400	10405 3413410	9589 2694537	4157 3413410	56.65 -26.68	3976 2806467	58.54 -4.15	3975 2806467	58.55 -4.15	3975 2713431	58.55 -0.70	
ind16	period area	2000	3077 5102874	3077 5092816	3067 5102874	0.32 -0.20	2916 4956462	5.23 2.68	2916 4956462	5.23 2.68	2913 4909999	5.33 3.59	
ind17	period area	1000	5613 5975345	5611 5968217	5612 5975345	-0.02 -0.12	5002 5942708	10.85 0.43	5002 5942708	10.85 0.43	4995 5898492	10.98 1.17	
average	period area					9.98 -2.71		15.89 1.07		16.37 1.07		17.33 1.69	

Table 2: Detailed results for performance and area optimization.

design	cells	PIs	POs	FFs	paths
s15850	2266	79	150	513	11582
s13207	1962	64	152	626	3146
s38584	6558	40	304	1274	13300
s38417	7407	30	106	1564	33232
s35932	8513	36	320	1728	4527
ind01	9182	132	64	69	4109
ind02	2545	94	297	367	3392
ind03	6668	71	103	560	10165
ind04	7639	897	337	1186	37145
ind05	7501	34	36	1472	111074
ind06	14454	148	63	2236	108269
ind07	32605	221	236	3661	2539280
ind08	54842	163	441	8856	3508497
ind09	74924	601	574	9254	2042374
ind10	68688	153	325	9316	4374454
ind11	170871	725	1037	11484	257018
ind12	108179	280	270	16688	238487
ind13	237173	317	273	16694	385666
ind14	292023	893	623	18332	2416761
ind15	264420	1901	1345	36641	12057648
ind16	282502	1095	611	38477	2514877
ind17	447231	2292	1692	86952	1562376

Table 1: Characteristics of the testcases.

the performance, the clock period (period), and the area obtained by a default synthesize run using the Cadence RTL Compiler. In the following columns, we show the results when using the Slack Balancing Algorithm (“clock”) and “synthesize”. The percentage improvements compare the results with the results of the standard flow (Column 5).

The Cadence RTL Compiler has the option to perform synthesis completely or incrementally. The latter consists only of a limited set of local operations to improve the timing and a global area recovery step, but it does not include technology independent optimization. For the results reported here we have chosen incremental synthesize. In general, complete synthesize achieved larger improvements in area, but smaller performance improvements.

Table 3 shows the run times of the Slack Balancing Algorithm (Algorithm 1). We give the target period, the initial worst slack, the worst slack with optimal latencies computed by the algorithm and finally the value up to which the slack was increased when the balance algorithm stopped (the slack of the edges of the last cycle found). In column eight we report how often a new edge becomes critical during the algorithm. The next column shows the number of cycles contracted, and the last column gives the CPU time in seconds. We contract up to 1000 cycles. Sometimes the algorithm encounters fewer cycles, for example for the first design, all nodes are contracted to a single node after only 362 cycles and

design	target period	slack zero latency	slack optimized	impr. (%)	slack balance stop	impr. (%)	edge changes	cycles contracted	cpu (sec)
s15850	750.00	-122.00	-113.50	0.97	2182.92	264.33	1100	362	0.30
s13207	600.00	-298.00	-284.00	1.56	820.00	124.50	1259	374	0.06
s38584	780.00	-125.00	-114.00	1.22	928.68	116.43	4280	998	0.35
s38417	820.00	-95.00	-84.00	1.20	539.00	69.29	3734	1000	0.50
s35932	400.00	-37.00	-37.00	0.00	133.00	38.90	3216	1000	0.17
ind01	7000.00	-4448.00	-781.50	32.03	7000.00	100.00	469	189	0.03
ind02	1500.00	-482.00	-400.50	4.11	1500.00	100.00	1242	460	0.14
ind03	2000.00	-2869.00	-2674.00	4.00	2662.94	113.62	1759	575	0.89
ind04	3000.00	-1828.00	-1667.00	3.33	-426.00	29.04	3449	1000	16.06
ind05	2300.00	-1054.00	-459.00	17.74	283.73	39.88	2669	1000	91.31
ind06	4020.00	-331.00	-290.00	0.94	545.50	20.14	3132	1000	301.57
ind07	2000.00	-3299.00	-3286.00	0.25	-3109.17	3.58	4460	300	1082.22
ind08	3000.00	-1208.00	-1207.00	0.02	-727.00	11.43	6607	1000	5990.19
ind09	5500.00	-18.00	899.00	16.62	1580.50	28.97	8008	1000	23.27
ind10	3000.00	-3011.00	-2984.00	0.45	-2323.60	11.44	9871	1000	11934.06
ind11	1.00	-908.00	-813.12	10.44	-712.95	21.46	14063	1000	173.35
ind12	3000.00	-1160.00	-1019.00	3.39	681.50	44.27	27199	1000	7.86
ind13	13200.00	-1879.00	-1820.00	0.39	-1374.50	3.35	11652	1000	89.42
ind14	1000.00	-5874.00	-4495.00	20.06	-3518.17	34.27	19315	1000	1613.38
ind15	13600.00	-7005.00	-757.00	30.32	-321.77	32.43	17747	1000	7609.06
ind16	2000.00	-1077.00	-1067.00	0.32	-738.75	10.99	41396	1000	1706.46
ind17	1000.00	-4613.00	-4612.00	0.02	-3861.50	13.39	6069	1000	12.22

Table 3: Runtime and slack improvements of the clock schedule algorithm

design		start	Initial Latency Shifts	Individual Latency Shifts	Incremental Synthesis	Synthesis
s15850	area	28864	28529	28335	28332	28578
	impr. (%)		1.16	1.83	1.84	0.99
	slack	0	0	3	3	0
s13207	area	28675	28264	28184	28173	27817
	impr. (%)		1.43	1.71	1.75	2.99
	slack	0	0	0	-1	0
s38584	area	71986	70683	70575	70539	71035
	impr. (%)		1.81	1.96	2.01	1.32
	slack	0	0	0	0	-10
s38417	area	90836	89391	88065	88011	91217
	impr. (%)		1.59	3.05	3.11	-0.42
	slack	0	0	0	0	-94
s35932	area	108764	108426	107567	107447	107850
	impr. (%)		0.31	1.10	1.21	0.84
	slack	0	0	0	0	0
ind05	area	207186	196184	196080	198712	195873
	impr. (%)		5.31	5.36	4.09	5.46
	slack	0	0	2	2	1
ind06	area	157564	156193	155941	155767	157721
	impr. (%)		0.87	1.03	1.14	-0.10
	slack	0	0	2	0	0

Table 4: Results Area Optimization.

the algorithm stops earlier.

The runtime of the Slack Balancing Algorithm is small compared to the runtime required by the Cadence RTL Compiler for “synthesize”, which is between 5 and 10 times higher.

As input for area optimization we use the output of the performance optimization. Table 4 shows the results. The columns give the total improvement achieved by the respective optimization phases shown in Algorithm 3 relative to the start values.

Table 4 shows that the algorithm always improved the area up to the *Incremental Synthesis* step. With initial shifts the area decreases fast, saving many individual shifts. The individual shifts improve the area steadily, but many loops are necessary to achieve a noticeable improvement. This is not surprising since one register does not influence the entire design, but only small parts of it. A final non-incremental synthesis improves the area infrequently.

6 Conclusions

We have presented algorithms for sequential performance and area optimization. Our experiments show that large improvements in performance and area can be achieved. The encouraging results motivate to direct further research on the realization of the dif-

ferent latencies for the registers, using a combination of retiming and clock tree construction. Our algorithm for performance optimization iteratively computes latencies, which optimally balance the slacks, and re-synthesizes the netlist. The algorithms for area optimization locally adjust the latencies to trade-off the area-delay sensitivities.

Acknowledgments

The authors would like to thank the Cadence RTL Compiler team, in particular David Seibert, Sascha Richter, and Adel Khouja for their active participation and rich input. They would like to thank Philip Chong and Ellen Sentovich for the constructive discussion during this project.

References

- [1] C. Leiserson and J. Saxe, “Optimizing synchronous systems,” *Journal of VLSI and Computer Systems*, vol. 1, pp. 41–67, January 1983.
- [2] C. Leiserson and J. Saxe, “Retiming synchronous circuitry,” *Algorithmica*, vol. 6, pp. 5–35, 1991.
- [3] J. P. Fishburn, “Clock skew optimization,” *IEEE Transactions on Computers*, vol. 39, pp. 945–951, July 1990.
- [4] 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*, vol. 10, pp. 74–84, January 1991.
- [5] I. S. Kourtev and E. G. Friedman, *Timing Optimization through Clock Skew Scheduling*. Boston, Dordrecht, London: Kluwer Academic Publisher, 2000.
- [6] C. Albrecht, B. Korte, J. Schietke, and J. Vygen, “Cycle time and slack optimization for VLSI-chips,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 232–238, November 1999.
- [7] S. Held, B. Korte, J. Maßberg, M. Ringe, and J. Vygen, “Clock scheduling and clocktree construction for high-performance ASICs,” in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (San Jose, California), pp. 232–239, November 2003.
- [8] C. Berman, D. Hathaway, A. LaPaugh, and L. Trevillyan, “Efficient techniques for timing correction,” in *IEEE International Symposium on Circuits and Systems*, pp. 415–419, May 1990.
- [9] K. Ravindran, A. Kuehlmann, and E. Sentovich, “Multi-domain clock skew scheduling,” in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (San Jose, California), pp. 801–808, November 2003.
- [10] A. Dasdan, S. S. Irani, and R. K. Gupta, “An experimental study of minimum mean cycle algorithms,” Tech. Rep. UCI-ICS 98-32, University of Illinois at Urbana-Champaign, 1998.
- [11] N. E. Young, R. E. Tarjan, and J. B. Orlin, “Faster parametric shortest path and minimum balance algorithms,” *Networks*, vol. 21, no. 2, pp. 205–221, 1991.
- [12] C. Albrecht, B. Korte, J. Schietke, and J. Vygen, “Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip,” in *Discrete Applied Mathematics*, vol. 123, pp. 103–127, November 2002.
- [13] S. M. Burns, *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, Pasadena, CA, December 1991.
- [14] R. Brodersen, M. Horowitz, D. Markovic, B. Nikolic, and V. Stojanovic, “Methods for true power minimization,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (San Jose, California), pp. 35–42, November 2002.