Efficient Incremental Clock Latency Scheduling for Large Circuits

Christoph Albrecht
Cadence Berkeley Labs, Berkeley, CA, USA

Abstract

The clock latency scheduling problem is usually solved on the sequential graph, also called register-to-register graph. In practice, the the extraction of the sequential graph for the given circuit is much more expensive than computing the clock latency schedule for the sequential graph. In this paper we present a new algorithm for clock latency scheduling which does not require the complete sequential graph as input. The new algorithm is based on the parametric shortest paths algorithm by Young, Tarjan and Orlin. It extracts the sequential timing graph only partly, that is in the critical regions, through a call back. It is still guaranteed that the algorithm finds the critical cycle and the minimum clock period. As additional input the algorithm only requires for every register the maximum delay of any outgoing combinational path. Computing these maximum delays for all the registers is equivalent to the timing analysis problem, hence they can be computed very efficiently. Computational results on recently released public benchmarks and industrial designs show that in average only 20.0 % of the edges in the sequential graph need to be extracted and this reduces the overall runtime to 5.8 %.

1. Introduction

The clock scheduling problem is usually solved on the *sequential graph*. The sequential graph has a node for every register and has a directed edge from a node u to a node v whenever there is a combinational path starting at the register corresponding to node u and ending at the register corresponding to node v. Associated with every edge is the maximum delay of all combinational paths from and to the respective registers. Generating the sequential graph is computationally expensive. The arrival times are propagated separately for each register over the complete fanout to the next registers. Every node and edge in the circuit graph is touched as often as it appears in the fanout of a register.

Simple timing analysis is performed on the *gate-level timing graph*, which has a node for every pin and an edge for every propagation segment within a gate as well as for every pair of a driver pin and a load pin connected by a net.

Though the number of nodes in the sequential graph is smaller compared to the gate-level timing graph, the number of edges can be enormous. Potentially, it can be n^2, where n is the number of nodes. Our computational results show that on practical instances the number of edges in the sequential graph can be up to 24 times larger than the number of edges in the gate-level timing graph.

It is computationally less expensive to determine for every register and primary input the maximum delay of any outgoing edge in the sequential graph. We only need to perform two longest paths delay propagations, forward for the arrival time computation and backward for the required arrival time computation. This is equivalent to the timing analysis problem and can be done in linear time. For a clock period T and a slack s at the output pin of the register the maximum delay of all outgoing paths of the register is given by $T - s$.

Our contribution is a new algorithm which does not require the complete sequential graph as input but which extracts parts of the sequential graph through a call-back mechanism. As additional input it requires for every node in the graph the maximum delay of all outgoing edges. As the algorithm optimizes the latencies it queries for a few selected nodes the outgoing edges through a call-back function.

Our algorithm is based on the parametric shortest paths algorithm by Young, Tarjan and Orlin [1] and it has the following differences: (1) Instead of one single shortest paths tree, an arborescence, we have several arborescences and at the beginning of the algorithm any single node forms an arborescence. This has the advantage that the latencies of most nodes are not changed. (2) During the algorithm by Young, Tarjan and Orlin the cost of the edges changes. The cost of each edge is a constant minus a single pareameter and this pareameter increases. The algorithm maintains a shortes path tree and in order to do so the algorithm needs to exchange edges in the tree, one edge at the time. This exchange step is the main step of the algorithm. The algorithm uses a heap to determine the next edge which needs to be exchanged. We add an additional step which queries the outgoing edges of a node based on the maximum delay specified at the beginning of the algorithm. We use a second heap to determine the next node for which the outgoing edges have to be queried. We show that the runtime of the algorithm is still $O(nm + n^2 \log n)$ as the original algorithm by Young, Tarjan and Orlin.

Our algorithm is especially useful when combined and iterated with other incremental optimization algorithms because it works only in the critical parts of the design. This becomes more and more important as the complexity of the optimization flows increases.

This paper is organized as follows: In Section 2 we discuss previous work, in Section 3 we give a formal definition of the clock latency scheduling problem, in Section 4 we describe the algorithms, in Section 5 we discuss extensions, and in Section 6 we present computational results.

3-9810801-0-6/DATE06 © 2006 EDAA

2. Previous Work

In the literature the clock latency scheduling problem is also known as the clock skew optimization problem. Fishburn [2] formulated the problem as linear program, and subsequently, further research on the clock latency scheduling problem was published ([3], [4], [5]).

Albrecht et al. [6, 7] and Kourtev et al. [8] consider the problem to not only compute a clock schedule which minimizes the clock period but which improves the slack on many other paths. For an overview of the clock latency scheduling problem, see Kourtev and Friedman [9] and Sapatnekar [10] (Chapter 9).

The problem of computing a clock schedule minimizing the clock period is equivalent to finding a minimum mean cycle in a directed graph. The minimum mean cycle is the cycle for which the average weight is minimum amoung all cycles, in a directed graph. Dasdan et al. [11] give an overview of the different algorithms for computing the minimum mean cycle in a directed graph and present experimental results about the performance of the algorithms for practical VLSI circuits. They conclude that Howard's algorithm is the fastest algorithm. However, no polynomial bound on the runtime for this algorithm is known. We could not see how this algorithm could be used for incremental clock latency scheduling. In practice the runtime to compute the minimum mean cycle is negligble compared to the runtime to extract the sequential graph.

To our best knowledge our work is the first work which computes the optimum clock period without the complete sequential graph.

3. Problem Formulation

Clock latency scheduling changes the latencies of the clock signal arriving at the registers. By increasing the latency of a clock signal at a register the slack of the incoming logic paths increases while the slack of the outgoing paths decreases. The opposite result is achieved by decreasing the latency.

For a given circuit the *sequential graph* is defined as follows: The nodes are the registers. A special host node represents all primary inputs and outputs. Whenever there is a combinational path from a register u to a register v, the graph contains an edge $e = (u, v)$. The weight $d(e)$ of this edge represents the maximum propagation delay of all paths from the register u to the register v. For a node v we denote by $l(v)$ the latency of the register corresponding to the node.

The problem to minimize the clock period T by clock latency scheduling can be formulated as the following linear program:

$$\begin{aligned} &\min\ T \\ &\text{s. t.}\quad l(u) + d((u,v)) - T \leq l(v) \quad \forall\,(u,v) \in E \end{aligned} \tag{1}$$

For a given feasible clock period T, the latencies can be determined by a longest path computation with the length of an edge e being $c(e) = d(e) - T$. It can be shown that the minimum clock period achievable by clock latency scheduling is equal to the maximum mean weight of all directed cycles in the sequential timing graph.

To simplify the presentation we ignore the setup time which could be added to the delay of the path. At the end of the paper we describe how further constraints, lower and upper bounds on the latencies and hold constraints, can be considered.

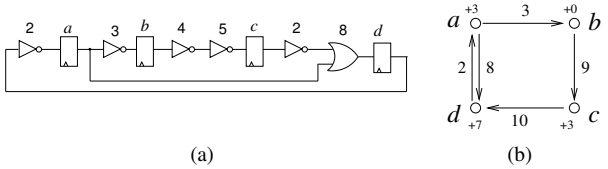

Figure 1. Circuit and the corresponding sequential graph.

Figure 1(a) shows a circuit with registers and gates. The numbers associated with the gates represent the delay. Figure 1(b) shows the corresponding sequential graph. Associated with the edges is the delay. Register a has a clock latency of +3, register b of +0, register c of +3 and register d of +7. This achieves the best possible clock period of 6.

4. Algorithms

In this section we present our clock latency scheduling algorithm to solve the linear program (1).

In difference to the algorithm by Young, Tarjan and Orlin we compute the maximum mean cycle instead of the minimum mean cycle, hence compute longest paths instead of shortest paths. To get the same behavior the edges could simply be multiplied by -1.

We present two algorithms, the first algorithm still requires the complete register-to-register graph as input, but maintains several arborescences instead of one single longest path tree as does the algorithm by Young, Tarjan and Orlin. The second algorithm incrementally extracts the sequential graph.

The algorithms use a heap which has the element with the largest key on the top. The function heap_init(H) initializes the heap H, heap_empty(H) returns true if the heap is empty, heap_add(H, x, k) adds an element x with the key k to the heap H, heap_key_max(H) returns the maximum key of all elements in the heap, heap_pop_max(H) returns the tuple of the element with the maximum key together with the key and removes this element from the heap, and heap_element(H, x) returns true if the element x is in the heap H.

In the following we describe the first algorithm, Algorithm 1. Contrary to the algorithm by Young, Tarjan and Orlin, we do not have a single directed tree, an arborescence, but a branching[1], that is several unconnected arborescences[2]. The edges of these arborescences are called critical edges, because they have zero slack. The algorithm keeps the critical edges in the set F. At the beginning of the algorithm the set of critical edges F is empty. Every node forms an arborescence by itself and is a root.

The algorithm maintains for each node v some variables: The total delay of the edges on the path from the root of the arbores-

1 A directed graph is called a *branching* if the graph does not contain a directed cycle and each node has at most one incoming (entering) edge.

2 A branching is an *arborescence* if the underlying undirected graph is connected. An arborescence has exactly one node without any incoming edge. This vertex is called the *root* of the arborescence.

Algorithm 1 Clock Latency Scheduling

Input: The sequential graph $G = (V, E, d)$.
Output: A critical cycle C, the minimum clock period T, and for each node $v \in V$ the latency $l(v)$ given by the function latency(v).

1: heap_init(H_{edge})
2: $F := \emptyset$
3: **for** all $v \in V$ **do**
4: set $\alpha(v) := 0$, $\beta(v) := 0$ and $\xi(v) := \emptyset$
5: **for** all $v \in V$ **do**
6: Let $e = (u, v)$ be the edge with edge_pivot$(e) = \max_{(u,v)\in E}$ edge_pivot$((u, v))$.
7: set $\mu(v) := e$ and $\eta(v) :=$ edge_pivot(e)
8: heap_insert$(H_{edge}, v, \eta(v))$
9: **while** (not heap_empty(H_{edge}) and heap_key_max$(H_{edge}) > -\infty$) **do**
10: set $(v, T) :=$ heap_pop_max(H_{edge})
11: set $e = (u, v) := \mu(v)$
12: **if** $(V, F \cup \{e\})$ contains a cycle C **then**
13: **return** (C, T)
14: set $F := F \setminus \xi(v) \cup \{e\}$ and $\xi(v) := e$
15: update_delays_and_levels(e)
16: update_edge_keys(v)
17: **return** $(\emptyset, -\infty)$

cence to the node is denoted by $\alpha(v)$ and the number of edges on this path is denoted by $\beta(v)$. We call $\beta(v)$ also the level of the node v. The critical incoming edge $e \in F$ of a node v is stored in $\xi(v)$, which might be empty ($\emptyset$). The variables are initialized in line 4.

During the algorithm, for the current clock period T, any path in F is always a longest path with respect to the length $d(e)-T$ for edge $e \in E$. This makes it possible to compute the latencies of a node v for the current clock period, which is returned by the function latency(v), that is $\alpha(v) - \beta(v)\,T$. At the beginning this property holds for a clock period T which is equal to or larger than the maximum delay of all edges. The clock period T decreases during the algorithm and the branching F is adjusted.

The algorithm maintains the heap H_{edge} which stores each node v with the clock period as key at which a new incoming edge (u, v) would become critical, if there were no other edges. This clock period is computed by the function edge_pivot$((u, v))$. At the beginning of the algorithm (lines 6–9) this clock period is simply equal to the delay of the edge, because $\alpha(v) = 0$ and $\beta(v) = 0$. The algorithm stores the next critical incoming edge of node v in the variable $\mu(v)$ and the clock period in $\eta(v)$. An edge (u, v) is critical if $l(u) + d((u, v)) - T = l(v)$. Since $l(v) = \alpha(v) - \beta(v)\,T$ for all $v \in V$, this is equivalent to

$$\alpha(u) - \beta(u)\,T + d((u, v)) - T = \alpha(v) - \beta(v)\,T.$$

Solving for T results in the expression in line 2 of Function 1, the function edge_pivot.

As long as the heap H_{edge} is not empty, the algorithm pops the next node v from the heap. If the new critical edge $e = (u, v)$ stored in $\mu(v)$ forms a cycle C with the already critical edges in F,

Function 1 edge_pivot$((u, v))$

1: **if** $(\beta(v) \le \beta(u))$ **then**
2: **return** $(\alpha(u) + d((u, v)) - \alpha(v))/(\beta(u) + 1 - \beta(v))$
3: **else**
4: **return** $-\infty$

Function 2 update_delays_and_levels$((u, v))$

1: Let $A = (V', F')$ be the maximal arborescence in G with root v and edges in F.
2: set $\Delta d := \alpha(u) + d((u, v)) - \alpha(v)$
3: set $\Delta l := \beta(u) + 1 - \beta(v)$
4: **for** all $w \in V'$ **do**
5: set $\alpha(w) := \alpha(w) + \Delta d$ and $\beta(w) := \beta(w) + \Delta l$

Function 3 update_edge_keys(v)

1: Let $A = (V', F')$ be the maximal arborescence in G with root v and edges in F.
2: **for** all $e = (u, v) \in E$ with $u \in V'$ and $v \notin V'$ **do**
3: **if** $(\eta(v) <$ edge_pivot$(e))$ **then**
4: set $\mu(v) := e$ and $\eta(v) :=$ edge_pivot(e)
5: heap_increase_key$(H_{edge}, v, \eta(v))$
6: **for** all $v \in V'$ **do**
7: Let $e = (u, v)$ be the edge with edge_pivot$(e) = \max_{(u,v)\in E}$ edge_pivot$((u, v))$
8: **if** $(\eta(v) >$ edge_pivot$(e))$ **then**
9: set $\mu(v) := e$ and $\eta(v) :=$ edge_pivot(e)
10: heap_decrease_key$(H_{edge}, v, \eta(v))$

Function 4 latency(v)

1: **return** $\alpha(v) - \beta(v)\,T$

then this cycle has the maximum mean delay and the algorithm returns the cycle with the current clock period (line 13). Otherwise, the arborescence is modified by a pivot step (line 14).

For each node w in the maximal arborescence $A = (V', F')$ in G with root v and edges in F the delay $\alpha(w)$ and the level $\beta(w)$ is updated by the function update_delays_and_levels$((u, v))$.

The reason that an edge $e = (u, v)$ becomes critical is that the path in F from one root to node u plus the edge (u, v) has more edges than the path in F from possibly another root to the node v. Hence the level $\beta(v)$ only increases during the algorithm.

As the delays and levels of the maximal arborescence in G with root v and edges in F are changed, the keys at which a new edge could become critical as well as the heap H_{edge} need to be updated by the function update_edge_keys(v). For an edge (u, v) with $u \in V'$ and $v \notin V'$ the pivot clock period can only increase (line 5), while for an edge (u, v) with $u \notin V'$ and $v \in V'$ the pivot clock period can only decrease (line 10). For all other edges the pivot clock period remains unchanged.

In practice, the algorithm terminates by finding a critical cycle (line 13) at a point at which most of the nodes still have a level of $\beta(v)$ equal to zero and these nodes then also get a latency of zero. The algorithm by Young, Tarjan and Orlin has a single arborescence and all nodes except for the single host node

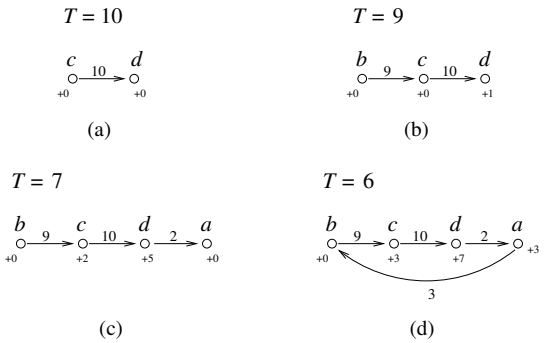

Figure 2. Snapshots of Algorithm 1 for the sequential graph in Figure 1(b).

have a critical incoming edge, a level different than zero and therefore most likely a latency different than one. The algorithm by Burns [3] is similar to our algorithm with respect to the property that it maintains several arborescences instead of one.

Figure 2 shows the different steps of the algorithm for the sequential graph in Figure 1(b). Shown are the critical edges in the branching F. The numbers at the nodes give the latency. The edge (c, d) is the first edge which becomes critical, then the edge (b, c), (d, a) and finally (a, b) which forms a directed cycle, the maximum mean cycle with an average delay of 6. This is the minimum clock period. In each of the figures the nodes are ordered with increasing level. For example, in Figure 2(c), the node b has level 0, the node c has level 1, the node d has level 2 and the node a has level 4.

We will now describe the incremental clock latency scheduling algorithm, Algorithm 2. The algorithm has one additional heap, the heap H_{node}. It is used to determine when the outgoing edges of a node need to be queried by the call-back function construct_out_edges. Since the latencies of the nodes only increase, an edge (u, v) can become critical if $l(u) + d^{out}(u) - T \geq 0$. With $l(u) = \alpha(u) - \beta(u)\,T$ this is equivalent to

$$\alpha(u) - \beta(u)\,T + d^{out}(v) - T \geq 0.$$

Solving for T gives

$$T \leq (\alpha(u) + d^{out}(u))/(\beta(u) + 1)$$

which is the expression in the function node_pivot(v), Function 5.

Once the outgoing edges of a node have been queried, the keys of the successor nodes are updated by the function update_successor_nodes. As before, the algorithm continuously decreases the clock period. It is ensured that whenever an outgoing edge of a node could become critical, it is extracted. When the branching F is updated (line 18) not only do the keys in the heap H_{edge} need to be updated, but also the keys in the heap H_{node} as done by the function update_node_keys (line 20). Young, Tarjan and Orlin use a Fibonacci-Heap and show that the runtime is $O(nm + n^2 \log n)$, where n is the number of nodes and m the number of edges. For a Fibonacci-Heap which has the element with the maximum key on the top the amortized time to increase a key is $O(1)$. Hence, the additional step to construct the outoing edges of the nodes including the update function update_successor_nodes has a total time of $O(m)$.

Algorithm 2 Incremental Clock Latency Scheduling

Input: The sequential graph $G = (V, E, d)$ given implicitly by the set of nodes V, the worst outgoing slack $d^{out}(u) = \max_{(u,v)\in E} d((u, v))$ for every node $u \in V$, and a function construct_out_edges(v), which generates the outgoing edges of the node v.

Output: A critical cycle C, the minimum clock period T, and for each node $v \in V$ the latency $l(v)$ given by the function latency(v).

```
1:  heap_init(H_edge)
2:  heap_init(H_node)
3:  F := ∅
4:  for all v ∈ V do
5:      set α(v) := 0, β(v) := 0 and ξ(v) := ∅
6:  for all v ∈ V do
7:      heap_insert(H_node, v,node_pivot(v))
8:  while (not heap_empty(H_node) and
            not heap_empty(H_edge)) do
9:      if (heap_max_key(H_node) > heap_max_key(H_edge))
        then
10:         set (v, T) := heap_pop_max(H_node)
11:         construct_out_edges(v)
12:         update_successor_nodes(v)
13:     else
14:         set (v, T) := heap_pop_max(H_edge)
15:         set e = (u, v) := μ(v)
16:         if (V, F ∪ {e}) contains a cycle C then
17:             return (C, T)
18:         set F := F \ ξ(v) ∪ {e} and ξ(v) := e
19:         update_delays_and_levels(v)
20:         update_node_keys(v)
21:         update_edge_keys(v)
22: return (∅, −∞)
```

Function 5 node_pivot(v)

```
1: return (d^out(v) + α(v))/(β(v) + 1)
```

Function 6 update_successor_nodes(u)

```
1: for all e = (u, v) ∈ E do
2:     if (η(v) < edge_pivot(e)) then
3:         set μ(v) := e and η(v) := edge_pivot(e)
4:         heap_increase_key(H, v, η(v))
```

Function 7 update_node_keys(v)

```
1: Let A = (V', F') be the maximal arborescence in G
   with root v and edges in F.
2: for all v ∈ V' do
3:     if (heap_element(H_node, v)) then
4:         heap_increase_key(H_node, v, node_pivot(v))
```

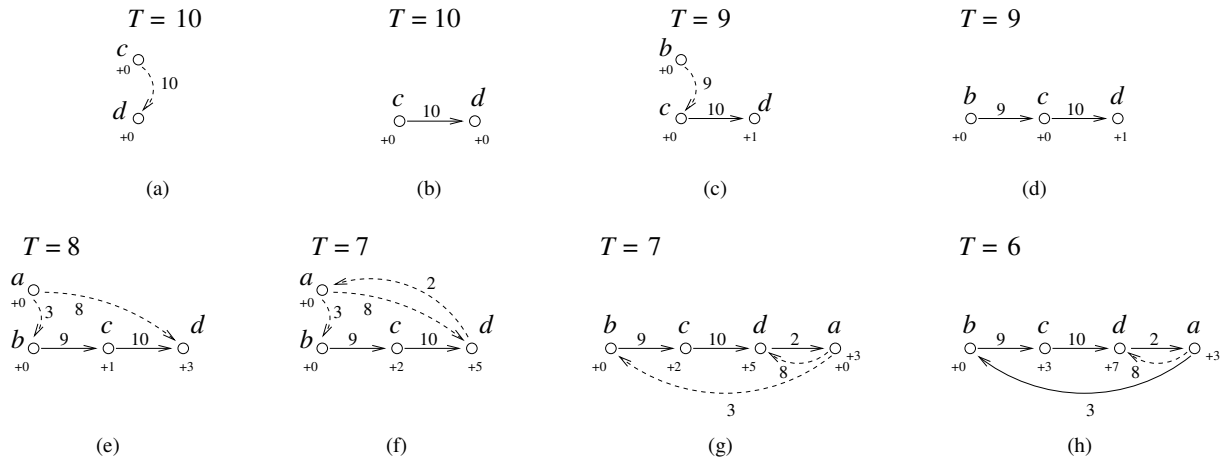

Figure 3. Snapshots of Algorithm 2 for the sequential graph in Figure 1(b).

Theorem 1 *The incremental clock latency scheduling algorithm, Algorithm 2, correctly finds the critical cycle and the minimum clock period in* $O(nm + n^2 \log n)$.

Figure 3 shows the different steps executed by the algorithm for the sequential graph in Figure 1(b). Shown are the critical edges in the branching F and dashed are the edges which have been extracted by calling the function construct_out_edges. In 3(a) this function has been called for node c, in 3(c) for node b, in 3(e) for node a and in 3(f) for node d. The Figures 3(b), 3(d), 3(g), and 3(h) correspond to the Figures 2(a-d). In each of these figures a new edge becomes critical. This example shows that not necessarily one of the newly extracted edges has to become critical in the next step.

5. Extensions

To keep the presentation of the algorithm simple we have only considered setup constraints and ignored hold constraints and other constraints, for example lower and upper bounds on the latencies of the registers. We could find the minimum clock period subject to the constraint that no hold constraints are violated. The algorithm by Young, Tarjan and Orlin can distinguish between parameterized edges and unparameterized edges. The setup edges are parameterized edges and the hold edges are unparameterized edges. These unparameterized edges are also used to implement lower and upper bounds on the latencies.

In order to extract only a subset of the hold edges we introduce a third heap and specify the minimum delay of all incoming edges for each node at the beginning. The algorithm extracts all incoming hold edges of a node as soon as the latency of the node becomes larger than the minimum delay of all incoming edges specified. Designs with multiple clocks with different clock periods can be optimized by using the slack of the combinational paths instead of the delay, rewriting (1), and maximizing the worst slack.

Finally, it is possible with the algorithm by Young, Tarjan and Orlin to not only optimize the worst slack, that is to find the most critical cycle, but to contract the cycle and to continue the algorithm, hence to increase the slack on other edges. For more details see [6, 7].

6. Computational Results

We have implemented the two algorithms presented and evaluated them within a commercial synthesis tool, Cadence RTL Compiler. To derive the outgoing edges in the sequential graph of a register, we increase the clock latency of the register to a high value, then query the slack at the data pins of the registers in the fanout and reset the clock latency of the register to zero again.

We use 9 designs of the recently released IWLS 2005 Benchmarks [12] and 8 industrial designs. Table 1 shows the characteristics of the testcases. Shown are the number of primary inputs (PIs), primary outputs (POs), registers (seqs), gates, pins and nets.

Table 2 presents the results achieved with the standard clock latency scheduling algorithm (Algorithm 1) and with the incremental clock latency scheduling algorithm (Algorithm 2). Shown are the number of nodes in the sequential graph, then the number of edges in the sequential graph and the runtime to extract the sequential graph plus the time to compute the maximum mean cycle (minimum clock period), then finally the number of edges extracted by the incremental clock latency scheduling algorithm and the runtime. Shown is also the percentage reduction in the number of edges in the sequential graph which have to be extracted and the percentage decrease in the runtime. For the runtime reported for Algorithm 1 in Column 4, 98.2 % is needed to extract to sequential graph and only 1.76 % for computing the maximum mean cycle. The runtime to extract one edge is certainly not unique for all the edges in the circuit and depends on the structure of the circuit. Hence, the reduction in the number of edges extracted may be smaller than the reduction of the runtime (ind01 and ind02) or larger (ind06 and ind07). In summary, our new incremental clock latency scheduling algorithm extracts only 20.01 % of the edges in the sequential graph (reduction of 79.99 %) and the runtime decreases by 94.23 % to only 5.77 %.

Acknowledgments

The author would like to thank his colleagues from the Cadence RTL Compiler team, especially Sascha Richter, for the close collaboration and a lot of help.

design	PIs	POs	seqs	gates	pins	nets
faraday_DSP	575	269	3629	33509	147984	32592
faraday_DMA	686	262	2202	15055	77939	13776
faraday_RISC	276	351	8051	61105	301749	60805
opencores_systemcaes	260	129	710	5701	30713	5157
opencores_mem_ctrl	115	152	1145	11501	71534	11175
opencores_pci_bridge32	162	207	3803	16485	93806	14915
opencores_ethernet	96	115	10545	41428	242823	40275
opencores_vga_lcd	89	109	17102	56522	359360	56234
gaisler_netcard	68	57	97873	342641	2119174	342265
ind01	162	229	14010	134874	822174	346658
ind02	2328	1692	87132	929820	5229168	2235468
ind03	223	236	3727	25885	196606	93181
ind04	166	441	9064	80297	380004	74376
ind05	156	325	9587	87555	631877	281066
ind06	422	475	3980	20275	163306	72935
ind07	215	230	12010	142452	1166099	525114
ind08	273	128	10554	114067	786693	350049
average	368	318	17360	124657	754177	268002

Table 1. Characteristics of the testcases.

		Clock Scheduling (Alg. 1)		Incr. Clock Scheduling (Alg. 2)			
design	nodes	edges	runtime (sec)	edges	%	runtime (sec)	%
faraday_DSP	3629	2039	28.00	192	-90.58	1.00	-96.43
faraday_DMA	2202	92090	12.00	33221	-63.93	4.00	-66.67
faraday_RISC	8051	319568	61.00	50916	-84.07	7.00	-88.52
opencores_systemcaes	710	32415	6.00	8206	-74.68	1.00	-83.33
opencores_mem_ctrl	1145	56405	14.00	30249	-46.37	8.00	-42.86
opencores_pci_bridge32	3803	92354	18.00	2735	-97.04	1.00	-94.44
opencores_ethernet	10545	250289	153.00	1659	-99.34	3.00	-98.04
opencores_vga_lcd	17102	616015	1490.00	549	-99.91	5.00	-99.66
gaisler_netcard	97873	19287654	17720.00	743530	-96.15	92.00	-99.48
ind01	14010	490378	1549.00	27525	-94.39	325.00	-79.02
ind02	87132	1633188	933.00	110100	-93.26	180.00	-80.71
ind03	3727	2573175	337.00	218547	-91.51	26.00	-92.28
ind04	9064	3405537	402.00	185622	-94.55	33.00	-91.79
ind05	9587	4220472	737.00	204085	-95.16	42.00	-94.30
ind06	3980	4267940	733.00	3586393	-15.97	320.00	-56.34
ind07	12010	4550388	1271.00	59694	-98.69	25.00	-98.03
ind08	10554	5331479	1372.00	4184349	-21.52	476.00	-65.31
average	17368	2777728	1578.59	555739	-79.99	91.12	-94.23

Table 2. Computational results for clock latency scheduling and incremental clock latency scheduling.

References

[1] 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.

[2] J. P. Fishburn, "Clock skew optimization," *IEEE Transactions on Computers*, vol. 39, pp. 945–951, July 1990.

[3] S. M. Burns, *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, Pasadena, CA, December 1991.

[4] N. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Graph algorithms for clock schedule optimization," in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (San Jose, CA), pp. 132–136, November 1992.

[5] R. B. Deokar and S. S. Sapatnekar, "A graph-theoretic approach to clock skew optimization," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 407–410, 1994.

[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/ACM International Conference on Computer-Aided Design*, (San Jose, CA), pp. 232–238, November 1999.

[7] 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.

[8] I. S. Kourtev and E. G. Friedman, "Clock skew scheduling for improved reliability via quadratic programming," in *Digest of Technical Papers of the IEEE/ACM International Conference on Computer-Aided Design*, (San Jose, CA), pp. 239–243, November 1999.

[9] I. S. Kourtev and E. G. Friedman, *Timing Optimization through Clock Skew Scheduling*. Boston, Dortrecht, London: Kluwer Academic Publisher, 2000.

[10] S. Sapatnekar, *Timing*. Norwell, MA: Kluwer Academic Publishers, 2004.

[11] 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.

[12] C. Albrecht, "IWLS 2005 Benchmarks," *International Workshop for Logic Synthesis (IWLS): http://www.iwls.org*, June 2005.