A Fast Incremental Clock Skew Scheduling Algorithm for Slack Optimization

Kui Wang, Hao Fang, Hu Xu and Xu Cheng
Department of Computer Science and Technology
Peking University
Beijing 100871, China
{wangkui,fanghao,xuhu,chengxu}@mprc.pku.edu.cn

Abstract—We propose a fast clock skew scheduling algorithm which minimizes clock period and enlarges the slacks of timing critical paths. To reduce the runtime of the timing analysis engine, our algorithm allows the sequential graph to be partly extracted. And the runtime of itself is almost linear to the size of the extracted sequential graph. Experimental results show its runtime is less than a minute for a design with more than ten thousands of flip-flops.

I. Introduction

Clock skew scheduling (CSS) is originally proposed to minimize the clock period[7, 6, 9, 12]. Later, its functionality is extended to slack optimization[13, 4, 8]. By increasing the slacks of critical paths, the permissible variations of clock arrival times are enlarged, which makes the chip more tolerant to process variations.

Currently, the best theoretical time bound for minimizing the clock period is $O(jm + j^2n)$[12], where m(respectively, n) is the number of arcs(respectively, nodes) in the graph and j is the maximum number of distinct edges in a shortest trail in the graph. In most cases, j can be regarded as a small constant, so the runtime is almost linear to the size of the graph. The best theoretical time bound for slack optimization is $O(nm + n^2 \log n)$[4].

Besides the runtime of CSS algorithms, the time for generating sequential graphs is also a noticeable overhead. The sequential graphs are generated by timing analysis engines as inputs for CSS algorithms. In the sequential graph, the paths between each pair of flip-flops are modeled as an arc and the maximum delay of the paths is the weight of the arc.

Recently, Albrecht[2] indicated that the extraction of sequential graph is time-consuming and implemented a CSS algorithm which allows the path delays to be extracted on demand. As additional input, the maximum delay of any outgoing combinational path from each flip-flop is calculated beforehand. And during the execution of the CSS algorithm, the necessary arc weights are got through callbacks to the timing analysis engine. Since many timing noncritical paths do not affect the CSS result, the weights of many arcs do not need extracting. Thus the runtime of timing analysis engine is reduced. Albrecht's algorithm does not consider the slack optimization problem. It only minimizes the clock period and the time complexity is $O(nm + n^2 \log n)$.

We propose a new algorithm for slack optimization. It has two important features which are only achieved in period minimization algorithms currently: near-linear time complexity and on demand path delay extraction.

In our algorithm, we use discrete values, i.e. integers, to measure the arc weights and clock delays. Using integers simplifies the design and analysis of our algorithm, but also introduce error. However, RC estimation and timing analysis also introduce error unavoidably, especially for pre-route designs. If we choose a small enough time unit, the error due todiscrete values will be ignorable compared to other unavoidable error.

After CSS, if the resulted clock delays are changed by smaller amounts, they are easier to be implemented by the clock tree synthesis tool. Our algorithm can be extended to reduce the total adjustment to the clock delays.

The rest of this paper is organized as follows. Section 2 gives formal definitions of the clock period minimization problem and the slack optimization problem. Section 3 describes our algorithm. Section 4 analyzes our algorithm theoretically. Section 5 discusses an extension to reduce the implementation cost. Section 6 presents experimental results. Section 7 concludes this paper.

II. Problem Formulation

A synchronous circuit with edge-triggered storage elements (flip-flops) is modeled as a sequential graph $G(V, E)$ which is strongly connected. In the sequential graph, each node represents a flip-flop and each arc represents the paths between a pair of flip-flops. Each flip-flop has its own clock delay, i.e., the propagation delay from the clock source to the clock pin of the flip-flop.

Due to space limitations, we ignore the hold time constraints related to the minimum delays of paths, though

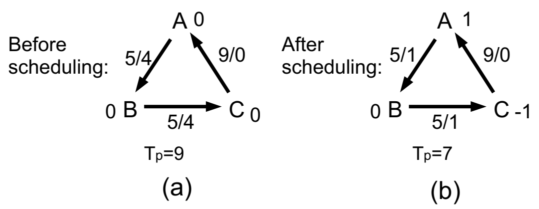


Fig. 1. Clock skew scheduling with discrete values

our algorithm can be easily extended to consider them. Thus only the maximum delays are associated with the arcs as their weights. The setup times of flip-flops are not modeled because they can be added to the delays of the paths. The slack of an arc is calculated as:

$$slack(e_{u,v}) = T_p - l(u) - w(e_{u,v}) + l(v)$$

where the T_p denotes the clock period, $w(e_{u,v})$ denotes the weight of the arc $e_{u,v}$ and $l(v)$ denotes the clock delay of the node v. In this paper, we assume they are integers, that is, integral times of a given delay unit.

The problem of minimizing the clock period T_p through CSS is formulated as finding an optimal clock schedule l such that:

$$\begin{aligned} &\forall e : slack(e) \geq 0 \qquad (1)\\ &T_p \rightarrow \min \end{aligned}$$

It can be proven that the minimum clock period achievable by CSS is equal to the mean weight of the *critical cycle* (also known as the *maximum mean cycle*, or MMC for short) in $G(V, E)$[10, 14]. When the optimal clock schedule l equalizes the slacks of the arcs in the MMC, the feasible clock period is minimized. So the clock period minimization problem is also referred as the MMC balancing problem.

When the path delays and clock delays are discrete values, the slacks of the arcs in the MMC can not be fully equalized. Fig. 1 illustrates this, where clock delay is marked near each node and *weight/slack* is marked near each arc. With clock delays measured with integers, T_p is minimized when there is such a cycle c:

$$\forall e \in c : \quad slack(e) \leq 1 \wedge \exists e \in c : slack(e) = 0 \qquad (2)$$

If the delays are measured with real numbers, T_p can be further reduced by no more than one delay unit. But this advancement is ignorable when the delay unit is small enough.

After T_p is minimized, clock delays can be further scheduled for slack optimization. By placing $l(v)$ at the middle of its feasible range, the slacks of the incoming paths and outgoing paths are balanced and thus the slacks of the critical paths are increased. In practice, $l(v)$ does not have to be at the middle when the slacks are large enough. So the slack optimization problem is formulated as finding an optimal clock schedule l for a given margin m such that:

$$\begin{aligned} &\forall v : |\min\{M(e_{u,v})\} - \min\{M(e_{v,w})\}| \leq 1 \qquad (3)\\ &\text{where } M(e) = \min(m, slack(e)) \end{aligned}$$

Because of the discrete delays, it may be impossible that the minimum slack of the incoming arcs equals that of the outgoing paths. So we just require that their difference is no larger than one delay unit.

```
Procedure BalanceMMC( $T_p$, $G(V,E)$ )
Input: $T_p$: the initial clock period.
       $G(V,E)$: the sequential graph.
Output: $T_{cycle}$: the minimized clock period.
        $G'$: the critical subgraph.
 1. while (true)
 2.     if ($\{e|slack(e) = 0\} = \emptyset$)
 3.         $T_p = T_p - 1$
 4.         calculate $\{w(e_{v,w})|T_p \geq W(v) \geq T_p - 1\}$
 5.         continue
 6.     //compute critical arcs of $G$ yielding the subgraph $G'$
 7.     $E' = \{e|slack(e) = 0 \vee slack(e) = 1\}$
 8.     $V' = \{v|\exists u : e_{u,v} \in E' \vee e_{v,u} \in E'\}$
 9.     $G' = (V', E')$
10.     $V'_r = \{v|v \in V' \wedge \forall u : e_{u,v} \notin E'\}$; $V'_h = V' - V'_r$
11.     if ($G'$ contains a cycle $c$)
12.         return $(T_p, G')$ //the MMC is balanced
13.     //compute the increment budget for each vertex
14.     $E_B = \{e_{v,x}|v \in V' \wedge x \notin V'\}$
15.     calculate $\{w(e)|e \in E_B\}$
16.     foreach $v \in V'_h$ in reversed topological order
17.       $\theta(v) = \min\{\{slack(e_{v,x}) - 1|e_{v,x} \in E_B\} \cup$
18.               $\{\max(0, \theta(w) + slack(e_{v,w}) - 1)|e_{v,w} \in E'\}\}$
19.     //increase the clock delay for each vertex
20.     foreach $v \in V'_r : \Delta(v) = 0$
21.     foreach $v \in V'_h$ in topological order
22.       $\Delta(v) = \min(\theta(v), \max\{\Delta(u) + 1 - slack(e_{u,v})|e_{u,v} \in E'\})$
23.     foreach $v \in V'_h : l(v) = l(v) + \Delta(v)$
```

Fig. 2. Pseudo code of our MMC balancing algorithm

III. Algorithms

We proposed an algorithm named extensive slack balancing[14] which can utilize a MMC balancing algorithm to solve the slack optimization problem. It is based on a simple observation: when the most critical circle is balanced, if we "isolate" it out of the graph, the second most critical circle will become the most critical one and get balanced.

We follow this approach. First we propose a MMC balancing algorithm, and then we give a refined version of the extensive slack balancing algorithm[14] and incorporate the MMC balancing algorithm into it.

Our MMC balancing algorithm is inspired by Burns' algorithm[3, 11], whose time complexity is $O(n^2m)$. However, the details are diverse because of the discrete delays and the need for on demand path delay extraction. The time complexity is also reduced. The basic idea is: taking the arcs whose slacks are 0 or 1 as the critical arcs, we incrementally change clock delays to increase the zero slacks until they are all ones. Then T_p is decreased by one, which changes the slacks to be 0 again. The algorithm iterates until the critical arcs construct at least one cycle.

Pseudo code of our MMC balancing algorithm is shown in Fig. 2. Line 2-5 ensures there is at least one zero-slack arc. Line 7-10 computes the critical subgraph which contains the critical arcs whose slacks are 0 or 1. The arcs with zero slacks will be optimized by increasing the clock delays of their heads. The arcs whose slacks are 1 will keep their slacks unchanged. V'_r and V'_h are the set of roots and the set of heads in the critical subgraph, respectively. Line 14-18 computes θ, the maximum possible increment for each head. E_B are the bordering arcs whose slacks are greater than one. They will sacrifice their slacks to the zero-slack arcs. θ can be viewed as budgets. The increment of clock delay to each node is not allowed to exceed its budget, so that no negative nor zero slacks are created in E_B or E'. Line 20-23 computes the real increment of each head according to its budget and the requirement of its proceeding critical arcs. To minimize the total increment, the clock delays are increased "just enough".

Like Albrecht's approach[2], the maximum delay of any outgoing combinational path from each flip-flop, which is denoted as $W(v)$ in Fig. 2, is calculated beforehand. During the iteration of *BalanceMMC*, the delays of individual arcs are calculated by the timing analysis engine when necessary (Line 4 and 15).

Fig. 3 illustrates one iteration of the loop body in *BalanceMMC*. Fig. 3(a) shows the initial status. The critical arcs are drawn in bold lines and the slacks are marked near arcs. In Fig. 3(b) and Fig. 3(c), the budgets and clock delays are calculated and marked near the nodes, respectively.

Although *BalanceMMC* can work fine with the original extensive slack balance algorithm, we choose to refine the original algorithm to eliminate its inefficiency and allow on demand path delay extraction. The refined version is shown in Fig. 4.

Firstly, when *BalanceMMC* terminates, there may be multiple cycles in G'. It is inefficient to discover and isolate the cycles one by one. So we just discover and isolate each nontrivial strongly connected component (SCC) in G'. Each SCC may have multiple cycles. The directed graph G' can be decomposed into its SCCs in $O(n+m)$ time[5].

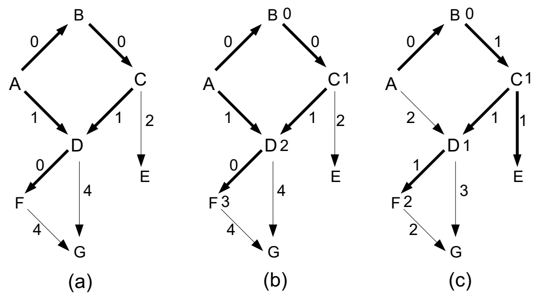


Fig. 3. An iteration of *BalanceMMC*

Secondly, before the original algorithm terminates, it undoes the topological and parametric changes which are made when isolating the MMCs. This is not necessary because a CSS algorithm outputs only the clock delays of the nodes in the original graph. In our refined version the changes are not undone and the clock delays are given by the procedure *ClockDelay* after the algorithm terminates.

Thirdly, we add to *IsolateSCC* the feature of on demand path delay extraction. Intuitively, *IsolateSCC* packs a SCC c into a single node n. The arcs connecting the nodes in c to other nodes are replaced with arcs connecting n. *IsolateSCC* invokes the timing analysis engine to ensure that the slacks of the new arcs are conservative (Line 6 and 10):

$$slack(e_{u,n}) = \min\{slack(e_{u,v}) | v \in V_c\} \quad (4)$$

$$slack(e_{n,w}) = \min\{slack(e_{v,w}) | v \in V_c\} \quad (5)$$

P maps each node in the isolated SCC to the new node n. It helps the calculation of the clock delays of the nodes in the original graph.

Fig. 5 shows the process of performing *ExtensiveSlackBalance* on a sequential graph. In Fig. 5, the clock delay is marked near each node and *weight/slack* is marked near each arc. And the critical arcs are drawn in bold lines.

IV. Theoretical Analysis

In this section we prove the correctness and effectiveness of our algorithms.

In *BalanceMMC*, only the computation of E' (Line 7) may need the weights of all arcs. To ensure on demand delay extraction is correct, we first prove E' can be correctly computed with only known arc weights.

Theorem 1. *During the progress of BalanceMMC, if the weight of an arc $e_{u,v}$ is unknown, then $slack(e_{u,v}) > 1$.*

Proof. If $w(e_{u,v}) \geq T_p - 1$, $w(e_{u,v})$ is known (line 4). If $l(u)$ is non-zero, $w(e_{u,v})$ is known (line 14-15). So if the weight of an arc $e_{u,v}$ is unknown, there are

$w(e_{u,v}) < T_p - 1$

$l(u) = 0$

Procedure *ExtensiveSlackBalance*($G(V,E)$, m, T_p)

Input: $G(V,E)$: the sequential graph. m: the margin for slack optimization. T_p: the initial clock period.

Output: l: the clock delay offset for each node. P: the map from a deleted node to the node which it is packed into.

```
1.  assert $T_p > \max\{W(v)\}$
2.  $T_S = T_p - m$
3.  foreach  $v \in V$: $P(v) = NIL$
4.  while (true)
5.      $(T_p, G')$=BalanceMMC($T_p$, $G$);
6.      if $(T_p \le T_S)$ return
7.      foreach scc in $G'$: $G$=IsolateSCC($G$, scc);
```

Procedure *IsolateSCC*(G, c)

Input: G: the sequential graph.
c: a strongly connected component in G.

Output: G: the changed sequential graph.

```
1.  create new node $n$; $l(n) = 0$
2.  $V_c = \{v|\exists e_{v,w} \in c\}$; $E_{del} = \{e_{u,v}|u \in V_c \lor v \in V_c\}$
3.  foreach  $v \in V_c$: $P(v) = n$
4.  foreach  $u \in V_s = \{u|\exists v : e_{u,v} \in E \land u \notin V_c \land v \in V_c\}$
5.      create arc $e_{u,n}$
6.      calculate $s = \min\{slack(e_{u,v})|v \in V_c\}$
7.      $w(e_{u,n}) = T_p - s - l(u)$
8.  foreach  $w \in V_e = \{w|\exists v : e_{v,w} \in E \land w \notin V_c \land v \in V_c\}$
9.      create arc $e_{n,w}$
10.     calculate $s = \min\{slack(e_{v,w})|v \in V_c\}$
11.     $w(e_{n,w}) = T_p - s + l(w)$
12. delete the arcs $E_{del}$ and the nodes $V_c$ from $G$
13. return  $G$
```

Procedure *ClockDelay*(v)

```
1.  $delay = l(v)$
2.  while $(P(v) \ne NIL)$
3.      $v = P(v)$; $delay = delay + l(v)$
4.  return  $delay$
```

Fig. 4. Pseudo code of the refined *ExtensiveSlackBalance*

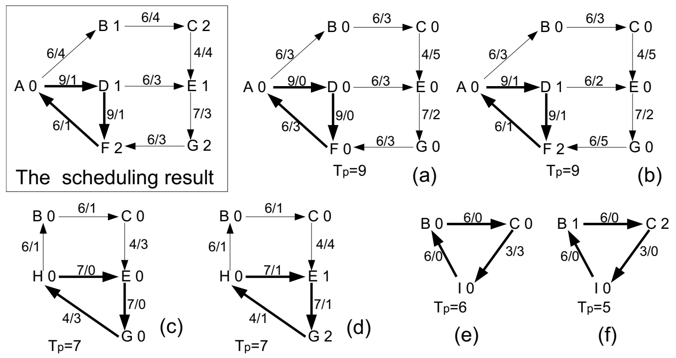
And *BalanceMMC* never decrease $l(v)$:
$l(v) \ge 0$
So we have
$slack(e_{u,v}) = T_p - l(u) - w(e_{u,v}) + l(v) > 1$

□

Then we prove *BalanceMMC* finds and balances the MMC eventually. Two lemmas should be proven beforehand.

Lemma 2. *After the increment of $l(v)$ in each iteration, the slacks of arcs in E' are not decreased, and the slacks of the arcs out of E' will be no smaller than one.*

Proof. If $e_{u,v} \notin E' \cup E_B$, then before increment $slack(e_{u,v}) > 1$ (line 7) and the clock delays of u and v are not changed, so after increment $slack(e_{u,v}) > 1$.



(a) the initial graph. (b) after *BalanceMMC*, clock delays of two nodes are adjusted and the MMC is A-D-F. (c) *IsolateSCC* packs A-D-F into vertex H. (d) after the second call of *BalanceMMC*, the MMC is H-E-G. (e) after the second call of *IsolateSCC*, the MMC H-E-G is packed into I. (f) after the third call of *BalanceMMC*.

Fig. 5. The progress of *ExtensiveSlackBalance*

If $e_{u,v} \in E_B$, then $\theta(u) \le slack(e) - 1$ (line 17) and $\Delta(u) \le \theta(u)$ (line 22), so after the increment of $l(u)$ (line 23), $slack(e) \ge 1$.

If $e_{u,v} \in E'$, then $\theta(u) \le \theta(v)$ (line 18) and thus $\Delta(u) \le \Delta(v)$ (line 22), so after the increment of $l(u)$ and $l(v)$, $slack(e)$ is not decreased.

□

Lemma 3. *The set of zero-slack heads, V_{S0}, is defined as $\{v|\exists u : slack(e_{u,v}) = 0\}$. And the level of a node v in V_{S0} is defined as the maximum number of the zero-slack heads on a trail in G' starting from v. At each iteration of BalanceMMC, if no MMC is found and returned, $L = \max\{level(v)|v \in V_{S0}\}$ is decreased by at least one.*

Proof. Let $V_f = \{v|v \in V' \land \forall w : e_{v,w} \notin E'\}$. No MMC is found, so $V_f \ne \emptyset$.

Because $\forall e \in E_B : slack(e) \ge 2$, we have

$$\min\{slack(e_{v,x}) - 1|e_{v,x} \in E_B\} \ge 1 \tag{6}$$

and thus

$$\forall v \in V_f : \theta(v) \ge 1 \tag{7}$$

Let z denote a zero-slack head whose level is 1. Let $e_{y,z}$ denote any critical path with zero-slack that ending at z. From (6) and (7), we know $\theta(z) \ge 1$, and thus:

$$0 \le \Delta(y) \le \theta(y) \le \theta(z) - 1$$
$$\Delta(z) = \min(\theta(z), \Delta(y) + 1) = \Delta(y) + 1 \tag{8}$$

Because of (8), $slack(e_{y,z})$ will be 1 after the increment of $l(y)$ and $l(z)$. And from Lemma 2 we know no zero-slack heads are created. So L is decreased by at least one. □

Theorem 4. *BalanceMMC terminates when it finds a strongly connected component c, which satisfies*

$$\forall e \in c : slack(e) = 0 \lor slack(e) = 1. \tag{9}$$

Proof. Because of Lemma 3, if the critical subgraph does not contain a cycle, after finite iterations, T_p is decreased by one. When the difference between T_p and the average weight of the arcs in some SCC is less than 1, the slacks of the arcs are 1 or 0 because $\forall e : slack(e) \geq 0$ (Lemma 2). Then at the next iteration, the SCC will be a part of the critical subgraph, which terminates *BalanceMMC*. □

It is worth noting that *BalanceMMC* may find a strongly connected component where the slacks are all ones when zero-slack arcs still exist. In other words, it may terminate when finding a cycle with arcs whose mean weight is not exactly the maximum. This penalty in precision is the cost of using discrete values. Fortunately, the zero-slack arcs will get optimized in the subsequent calls of *BalanceMMC*.

Next we will prove *ExtensiveSlackBalance* solves the slack optimization problem.

Theorem 5. *When ExtensiveSlackBalancing terminates, (3) holds if the slacks are calculated with the initial value of T_p.*

Proof. When it terminates, for a existing node u, there is

$$slack(e_{x,u}) \geq m \wedge slack(e_{u,y}) \geq m$$

So (3) holds.

For a deleted node v, before the invoking of the *IsolateSCC* which deletes it, it belongs to a cycle. Due to Theorem 4, (3) holds when v is deleted. Due to (4) and (5), (3) holds for the initial graph.

□

Theorem 6. *The complexity of ExtensiveSlackBalance is $O(kT_pm')$. k represents the maximum possible value of $L = \max\{level(v)|v \in V_{S0}\}$ for any critical subgraph G' during the execution. m' denotes the number of the edges which is ever in the critical subgraph.*

Proof. We say an iteration of the loop in *BalanceMMC* is trivial when it returns T_p. Because of Lemma 3, if no cycle is found, after k nontrivial iterations of the loop in *BalanceMMC*, T_p is decreased by one. And trivial iterations are less than nontrivial iterations because *IsolateSCC* removes all the cycles. So the total iteration count is $O(2k(T_p - T_S)) \sim O(kT_p)$.

The cost of each iteration of the loop in *BalanceMMC* is $O(m')$. The cost of all the invokings of *IsolateSCC* is also $O(m')$. So the total time complexity of *ExtensiveSlackBalance* is $O(kT_pm')$.

□

In real circuits, T_p has bounded values and in our experiments k is no larger than 8. We believe k is a small number in most cases. So the runtime of our algorithm is almost linear to the size of the extracted critical subgraph.

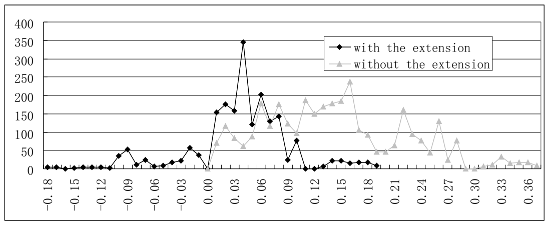


Fig. 6. The distribution curve of non-zero clock delays (slack optimization for the b19 testcase)

V. Extension

In most cases, the clock delays of many flip-flops keep zero after CSS because many paths are not timing critical. A clock schedule which makes small adjustments to the clock delays is easier to implement because the clock delays are close to zero. The solution of the slack optimization problem is not unique. A solution with small adjustments is preferred. The sum of difference between the obtained clock delay after CSS and the target clock delay for every flip-flop is defined as the cost of CSS[15]. We use zero as the target clock delay in this paper, so the cost is $\sum |l(v)|$.

BalanceMMC is not cost-efficient. It only increases the clock delays and never decreases them, so the slacks of the noncritical arcs which end at the nodes in G' are not utilized. If *BalanceMMC* takes advantage of both increment and decrement, it can generate a clock schedule with smaller cost.

In fact, the loop body of *BalanceMMC* can be easily modified to be decrement-only: in its loop body just *reversely* interpret the direction of each arc and the sign of each clock delay. So if *BalanceMMC* executes its original loop body and this reversed version interleavingly, it can utilize the slacks at both directions. Our experiments show that this extension can reduce the cost remarkably (see Fig. 6).

VI. Experimental Results

We choose seven testcases from the IWLS 2005 benchmarks[1]. Table I shows their characteristics. Our algorithm is written in Java and runs on a 2.0GHz Opteron processor. The Burns' algorithm[3, 11], our algorithm and its extension are used to solve the MMC balancing problem and the slack optimization problem for these testcases. The delay unit is 5ps, which is reasonably small considering that the timing differences between pre-layout and post-layout designs are usually tens of picoseconds. The results are shown in Table II.

The full sequential graphs are extracted by Synopsys PrimeTime. Currently we query arc weights from the sequential graphs to emulate the callbacks to PrimeTime. The presented runtime does not include the time for tim-

TABLE II
THE RESULTS OF EXPERIMENTS

id	MMC balancing										Slack optimization ($m = T_{MAX} - T_{MMC} + 100ps$)									
	Burns		ours				ours_ext				Burns		ours				ours_ext			
	rt	cost	rt	%arc	cost	k	rt	%arc	cost	k	rt	cost	rt	%arc	cost	k	rt	%arc	cost	k
1	38.8	5.1	0.7	31.4	4.1	3	0.6	42.0	2.0	2	165.4	187.3	17.0	76.5	204.0	6	11.1	92.7	30.5	6
2	50.3	4.0	0.7	7.4	2.7	2	0.8	13.0	1.0	1	358.9	338.7	30.5	67.8	316.8	5	20.1	83.8	95.4	5
3	422.6	64.2	5.1	29.8	59.4	2	4.8	44.3	28.2	4	956.1	463.1	64.9	50.1	477.3	7	46.9	58.7	106.1	4
4	16.6	53.5	8.8	6.0	45.9	2	8.5	12.9	14.7	2	95.8	778.3	17.1	24.0	748.8	3	14.6	86.8	228.0	3
5	3.9	1.2	0.1	4.5	0.01	1	0.1	9.5	0.01	1	29.7	373.6	4.2	36.2	353.0	4	3.9	93.6	190.2	7
6	3.6	22.3	0.3	42.0	22.4	2	0.3	62.0	7.6	2	10.13	285.0	2.2	67.1	274.2	8	1.2	90.0	42.7	5
7	39.8	8.9	0.3	11.9	3.4	2	0.2	7.2	0.01	2	124.3	311.8	10.5	80.1	299.0	7	8.3	65.5	65.6	7

Burns: the Burns' algorithm ours: our algorithm ours_ext: our algorithm with the extension
rt: runtime of the algorithm (in second) %arc: the percentage of the arcs whose weights are queried.
cost: the sum of the obsolete values of the clock delays (in ns)

TABLE I
THE CHARACTERISTICS OF TESTCASES

id	name	#nodes	#arcs	Ext	T_{MAX}	T_{MMC}
1	b17	1432	167077	746	1968	1951
2	b18	3306	395432	2818	2783	2765
3	b19	6604	791300	5034	3356	3290
4	ethernet	10580	252306	813	2223	1655
5	pci_bridge	3427	83086	118	1367	1315
6	usb_fuct	1785	26374	26	1353	1283
7	DMA	2279	129089	347	1378	1369

Ext: runtime of the extraction of the sequential graph (in second)
T_{MAX}: maximum delay of all the paths (in ps)
T_{MMC}: maximum mean delay of the cycles (in ps)

ing analysis. Instead, we use the percentage of the arcs which have ever been queried to represent the runtime for timing analysis.

The runtime of our algorithm is much shorter than the time for timing analysis. The maximum of the parameter k in the experiments is only 8. The experiments also show k is independent of the circuit size, so we believe in larger circuits, it is still a small number. This strengthens our confidence of the near-linear time complexity of our algorithm. Thus for larger circuits, its runtime is still expected to be shorter than that for timing analysis.

When the decrement of T_p is small, that is, the difference between T_{MAX} and T_{MMC} is small, the reduction of the runtime in path delay extraction is remarkable because only a small portion of the arc weights need extracting. When the decrement of T_p is large, almost all the arc weights are extracted, so the reduction in runtime is limited.

The extension of our algorithm reduces the cost dramatically. However, it needs more arc weights extracted to work.

VII. Conclusion

We propose a novel clock skew scheduling algorithm for clock period minimization and slack optimization. Its advantages are three folds: (1) it is fast in theory (almost linear) and in practice, (2) it allows on demand delay extraction, (3) it both increases and decreases the clock delays to reduce the cost.

References

[1] C. Albrecht. IWLS 2005 benchmarks. *International Workshop for Logic Synthesis (IWLS)*, June 2005.

[2] C. Albrecht. Efficient incremental clock latency scheduling for large circuits. *Proceedings of Design, Automation and Test in Europe*, pages 1091 – 1096, 2006.

[3] S. M. Burns. Performance analysis and optimization of asynchronous circuits. *PhD thesis, California Institute of Technology, Pasadena, CA*, 1991.

[4] C.Albrecht, B.Korte, J.Schietke, and J.Vygen. Cycle time and slack optimization for VLSI-chips. *Digest of Technical Papers of the International Conference on Computer-Aided Design*, pages 232–238, 1999.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to algorithms. *MIT Press, Cambridge, Mass*, 1991.

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

[7] J. P. Fishburn. Clock skew optimization. *IEEE Transactions on Computers*, 39(7):945–951, July 1990.

[8] I. S. Kourtev and E. G. Friedman. Clock skew scheduling for improved reliability via quadratic programming. *Digest of Technical Papers of the International Conference on Computer-Aided Design*, pages 239–243, 1999.

[9] J. L. Neves and E. G. Friedman. Optimal clock skew scheduling tolerant to process variations. *Proceedings of the 33rd Design Automation Comference*, pages 623–628, 1996.

[10] M. C. Papaefthymiou. Understanding retiming through maximum average-delay cycles. *Proceedings of the 3rd ACM symposium on Parallel algorithms and architectures*, pages 65–84, 1994.

[11] K. Ravindran, A. Kuehlmann, and E. Sentovich. Multi-domain clock skew scheduling. *Digest of Technical Papers of the International Conference on Computer-Aided Design*, pages 801–808, 2003.

[12] A. Takahashi. Practical fast clock-schedule design algorithms. *IEICE Transaction on Fundamentals*, E89-A(4):1005–1011, April 2006.

[13] A. Takahashi and Y. Kajitani. Peformance and reliability driven clock scheduling of sequential logic circuits. *Proceedings of Asia and South Pacific Design Automation Conference*, pages 37–42, 1997.

[14] K. Wang, L. Duan, and X. Cheng. ExtensiveSlackBalance: an approach to make front-end tools aware of clock skew scheduling. *Proceedings of the 43rd Design Automation Comference*, pages 951– 954, 2006.

[15] T. Yoda and A. Takahashi. Clock schedule design for minimum realization cost. *IEICE Transaction on Fundamentals*, E83-A(12):2552–2557, April 2000.