A Fast, Iterative Clock Skew Scheduling Algorithm with Dynamic Sequential Graph Extraction

Shijian Chen[1,2,3], Yihang Qiu[3], Biwei Xie[1,2,3], Mingyu Chen[1,3], Xingquan Li[2,✉]
[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]Pengcheng Laboratory, Shenzhen, China
[3]University of Chinese Academy of Sciences, Beijing, China
{chenshijian22, qiuyihang23}@mails.ucas.ac.cn, {xiebiwei, cmy}@ict.ac.cn, lixq01@pcl.ac.cn

***Abstract*—Clock skew scheduling (CSS) is a well-known technique that improves design timing slack by adjusting clock latency to flip-flops. CSS requires obtaining timing path information between sequential elements (including flip-flops and I/O ports), known as sequential graph extraction, which is the most time-consuming part of advanced CSS. In this paper, to quickly identify the potential of clock skew in slack optimization, we propose an iterative CSS algorithm that leverages timing propagation to facilitate sequential graph extraction. Then, we provide a comprehensive skew calculation method that considers multiple clock latency constraints, obtaining the target latency of each flip-flop. Finally, we present slack optimization techniques to achieve the target latencies. Our algorithm achieves a 49.11× speedup compared to the advanced CSS algorithm based on partial graph extraction, reducing 90.05% of the extracted edges. Compared to a state-of-the-art CSS-based slack optimization methodology, our algorithm delivers a 27.01× speedup with superior slack improvement.**

I. Introduction

Slack improvement is critical for achieving timing closure, and clock skew scheduling (CSS) is a well-known technique that enhances design timing slack by adjusting clock latency to flip-flops [1], also called useful skew optimization. Previously, Chan et al. proposed NOLO ("no-loop") [2], which applied CSS during post-synthesis to determine predictive skew. This predictive skew facilitated slack optimization, reducing runtime by 66% and improving total negative slack (TNS) by 5%. Similarly, Kim et al. [3] introduced the Fast Predictive Useful Skew Methodology (FPM), which applied CSS to compute predictive skew during placement, leading to significant speedups and improvements in negative slack. These findings highlight the significant potential of CSS in slack optimization.

In fact, CSS has been well-studied [4] [5] [6] [7]. Graph-based CSS [8] is the dominant approach, which uses a parametric shortest path algorithm to find the maximum mean weight cycle (MMWC). The time complexity of the MMWC solution has been proven as $O(nm+n^2 \log n)$, where n and m represent the vertices and edges of the sequential graph[1], respectively. In practice, the MMWC solution achieves significant timing slack improvements and is currently the standard approach for useful skew optimization in commercial EDA tools [2]. However, Albrecht [9] noted that the runtime required to compute the MMWC is negligible compared to the time required to extract the sequential graph from the gate-level timing graph[2]. Consequently, Albrecht [9] first introduced incremental clock skew scheduling (IC-CSS), demonstrating that extracting only 20% of the edges in the sequential graph is sufficient to identify the maximum mean cycle, reducing overall runtime to 5.8%. Subsequently, Wang [10] extended the MMWC solution for slack optimization and proposed an effective cycle-handling method. Notably, Wang continued to employ the same edge extraction method as IC-CSS, and the runtime reported in the study does not account for the time spent on edge extraction.

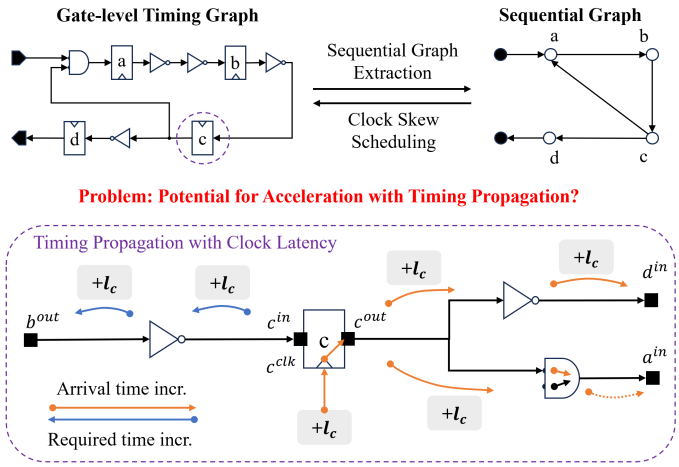

Fig. 1: Default clock skew scheduling flow: (i) Sequential graph extraction, (ii) Clock skew scheduling on the sequential graph. - Inspiring acceleration by leveraging timing propagation.

Despite advancements in CSS and efforts in IC-CSS, two challenges remain. First, we observe that the sequential graph extraction in IC-CSS still retrieves many non-essential edges, indicating that the extraction runtime can be further optimized. Section III provides the details on this issue. Second, no further studies have explored CSS under clock latency constraints. In fact, considering these constraints inevitably requires extracting additional sequential edges, and solving for the MMWC under such conditions may not always be feasible.

Fig. 1 illustrates the key challenges in CSS that our work aims to address. Recently, many static timing analysis (STA) timers have been open-sourced [11] [12] [13], enabling researchers to utilize these timers effectively for slack optimization. Additionally, significant progress has been made in accelerating timers [14] [15] [16] [17] [18]. Since the sequential graph is a subgraph of the gate-level timing graph, this motivates the use of timing propagation to improve CSS.

In this work, to quickly identify the potential for useful skew in slack optimization, we propose an iterative clock skew scheduling algorithm with dynamic sequential graph extraction inspired by timing propagation. The key contributions are summarized as follows.

- We propose a fast iterative clock skew scheduling algorithm that extracts only the essential sequential graph for skew calculation.
- We design a comprehensive skew calculation method based on the essential sequential graph, guaranteeing slack enhancement in each iteration while considering multiple latency constraints.
- We apply reconnection and cell movement techniques to achieve the desired latency.
- Compared to the modified IC-CSS method, our iterative sequential graph extraction reduces the search for essential edges and achieves a shorter runtime. Compared to state-of-the-art CSS-

This work is supported in part by the Major Key Project of PCL (No. PCL2023A03), the National Natural Science Foundation of China (No. 62090021), the Natural Science Foundation of Fujian Province (No. 2024J09045).

[1]Sequential graph: a directed graph with flip-flops or I/O ports as vertices and timing paths as edges.

[2]Gate-level timing graph: a graph used in timing propagation, where vertices represent gates or pins, and edges represent timing arcs or nets.

based slack optimization methodologies, we achieve notable runtime reductions and improved slack.

II. Slack Improvement via Clock Skew Scheduling

A. Problem Formulation

There are two types of timing slack: early slack and late slack.

$$s^E_{u,v} = l_u - l_v + t^{c2q}_u - t^{c2q}_v + d^{min}_{u,v} - t^{hold}_v \tag{1}$$

$$s^L_{u,v} = l_v - l_u + T - t^{c2q}_u - d^{max}_{u,v} - t^{setup}_v \tag{2}$$

Here, $s^E_{u,v}$ and $s^L_{u,v}$ represent the early and late slack on the timing path from u to v, l_u and l_v denote clock latencies, t^{c2q}_u and t^{c2q}_v are the clock-to-Q delays, $d^{min}_{u,v}$ and $d^{max}_{u,v}$ represent the minimum and maximum path delays, T is the given clock period, t^{hold}_v and t^{setup}_v are the hold time and setup time of the flip-flop node v.

Focusing on evaluating and applying CSS to incremental slack optimization, we treat all variables in Eq. (1) and Eq. (2), except clock latencies, as constants. The relationship between slack variation and clock latencies is expressed as follows:

$$\Delta s^E_{u,v} = (l_u - l_v), \quad \Delta s^L_{u,v} = (l_v - l_u) \tag{3}$$

In the following discussion, we use $(l_v - l_u)$ to represent the skew increment for the flip-flop node from u to v.

Define a slack sequence $s = \{s_1, s_2, \ldots, s_{2n}\}$ that includes all early and late slack variables, where n is the number of timing paths. The sequence s is defined to satisfy the following conditions:

- The slack variables are in non-decreasing order, expressed as $s_1 \leq s_2 \leq \cdots \leq s_{2n}$.
- When slack > 0, set slack $= 0$.

We impose constraints of all edges E' on clock skew:

$$l_u - s^L_{u,v} \leq l_v, \text{ and } l_u + s^E_{u,v} \geq l_v, \quad \forall e_{u,v} \in E' \tag{4}$$

We impose constraints on the upper and lower bounds of latency:

$$l^{min}_v \leq l_v \leq l^{max}_v, \quad \forall v \in V \tag{5}$$

Our slack optimization problem consists of finding a solution of Eq. (4) and Eq. (5) such that the sequence s is lexicographically maximal. For example, between $s = \{-5, -4, -2\}$ and $s' = \{-5, -3, -3\}$, we say s' is greater than s because their second elements differ ($-3 > -4$). Since only $s < 0$ requires optimization, it is also referred to as the negative slack optimization (NSO) problem of clock skew scheduling.

Then, we construct a directed sequential graph instance $G = (V, E', w)$ for our CSS problem. Here, V consists of all flip-flop nodes along with two supernodes representing the input and output ports. The timing paths between all nodes form the edge set E' of G, and w represents the edge weights, which correspond to the negative slacks. We direct the late edges from the launch flip-flop to the receive flip-flop, assigning the late slack s^L as the edge weight w. Conversely, we direct early edges from the receive flip-flop back to the launch flip-flop, using the early slack s^E as the edge weight w. By setting the edges in different directions, we incorporate $\Delta s^E_{u,v}$ and $\Delta s^L_{u,v}$ from Eq. (3) into graph G. Finally, we define the weight of each vertex in graph G.

$$w^{out}_u = \min_{e_{u,v} \in E'} w_{u,v}, \quad \forall u \in V \tag{6}$$

To clarify skew calculation, we define the concept of an arborescence and its associated properties. An arborescence is characterized by the following properties: It is a directed acyclic graph, and except for the root vertex, which has no incoming edges, every other vertex has exactly one incoming edge.

We refer to the path from the root to a node of the arborescence as a *path* and define functions for the vertices along it.

$$\alpha(v) = \sum_{e_{u,v} \in P_{\text{root}\to v}} w_{u,v}, \text{ and } \beta(v) = |P_{\text{root}\to v}| \tag{7}$$

where $P_{\text{root}\to v}$ is the path from root to vertex v.

B. Preliminaries of Incremental Clock Skew Scheudling

By extracting only partial edges from the sequential graph, Albrecht [9] achieves incremental clock skew scheduling (IC-CSS), significantly reducing runtime. The key to partial sequential graph extraction is that IC-CSS calculates the maximum delay of outgoing timing paths only once, denoted as d^{out}. When a node u satisfies Eq. (8), it is considered as a critical vertex and has potential critical edges in the outgoing direction:

$$l_u + d^{out}_u - T \geq 0 \tag{8}$$

where l_u is the latency increment and T is the period variable defined in IC-CSS. Since the goal of IC-CSS is to find the minimum period T, it iteratively reduces T. When a critical vertex u satisfying Eq. (8) is identified, it invokes a call-back mechanism to extract all outgoing edges of u. The algorithm terminates when encountering a cycle, which is also referred to as the maximum mean cycle. As not all vertices are critical and do not need their outgoing edges retrieved, IC-CSS completes clock skew scheduling using only a partial sequential graph. We refer to the T-variable updates and latency calculations in IC-CSS as shown in Eq. (9).

$$T = \alpha(v)/\beta(v), \text{ and } l_v = \alpha(v) - \beta(v) \cdot T \tag{9}$$

The detailed process is provided in Alg. 2 of [9].

III. Our Algorithm

A. Exploring Further Acceleration on Sequential Graph Extraction

IC-CSS demonstrates that minimizing the cycle period in CSS does not require extracting the complete sequential graph. To maintain the global minimum cycle period, IC-CSS must extract all outgoing edges for critical vertices to find MMWC, as discussed in Section II-B. Many of these outgoing edges remain unused, presenting opportunities for further acceleration. Moreover, IC-CSS does not incorporate a CSS method under latency constraints, which can impact the identification of the MMWC and require extracting additional edges.

To rapidly solve the proposed NSO problem using clock skew, we circumvent the need to maintain the MMWC, which enables faster batch extraction of all essential sequential edges. Since the STA timer quickly identifies negative (essential) edges in a sequential graph, which corresponds to reporting timing slack violations, we use it to extract and update essential sequential edges. Updating essential edges is achieved by first controlling the clock latency of flip-flops, followed by timing propagation. Next, we further explain the iterative process of extracting a sequential graph and the method for calculating skew within the partial sequential graph, considering latency constraints.

B. Iterative Sequential Graph Construction

If an edge in the sequential graph consistently satisfies $s \geq 0$, it does not require CSS for negative slack optimization. Such edges can be omitted from extraction, thereby reducing runtime. We refer to edges required for skew optimization as **Essential Edges**, while the others are classified as **Non-Essential Edges**.

Fig. 2 shows the core idea of our iterative sequential graph extraction, compared to the sequential graph extraction approach used in IC-CSS. As observed in Fig. 2, the call-back mechanism of IC-CSS extracts many non-essential edges. To extract only essential edges for CSS, we utilize an STA timer to track changes in essential edges and extract them in each iteration, a process known as the Update-Extract Mechanism. We propose a comprehensive process as follows:

i. Extract the partial sequential graph of essential edges.
ii. Generate multiple arborescences based on the partial sequential graph and perform skew calculation.
iii. Add the clock latency values from skew calculation to corresponding flip-flop nodes, perform timing propagation, and repeat the iterative sequential graph construction process.

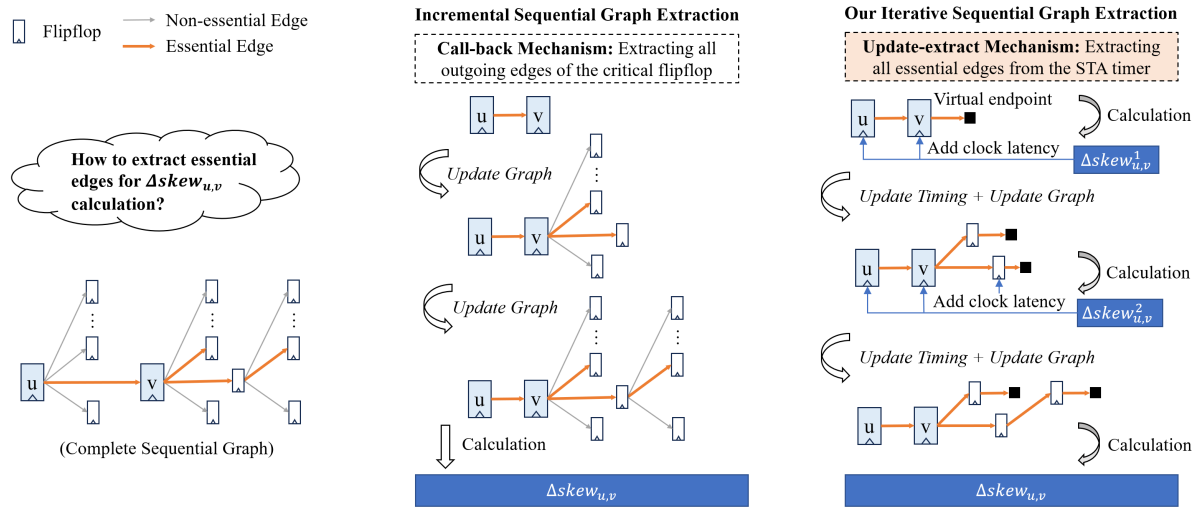

Fig. 2: Comparison of incremental sequential graph extraction from [9] and our iterative sequential graph extraction.

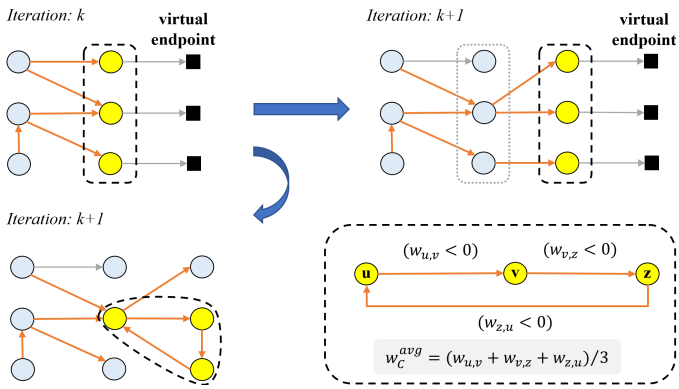

Fig. 3: A snapshot from the iterative sequential graph construction process. The dashed box highlights the vertices that need processing: adding virtual endpoint connections or addressing a cycle.

Finally, we demonstrate that the proposed algorithm reduces the time complexity of sequential graph extraction from nm' to km', where n is the number of vertices, m' is the number of nets in the gate-level timing graph, and k is the number of iterations. Generally, k is much smaller than n.

1) Update-Extract Mechanism

We propose the Update-Extract Mechanism to iteratively extract the sequential graph, which is crucial for avoiding the extraction of non-essential edges. It first updates all computed clock latency values for the clock inputs of the selected flip-flops, followed by timing propagation to update the slack for the relevant timing paths. Finally, the violation timing paths will be extracted as essential edges. In practice, we focus on the newly violated timing endpoints to minimize redundant searches for essential edges and identify new essential edges from these endpoints. The update method for the remaining essential edges is shown in Eq. (10).

$$w^{k+1}_{u,v} = w^k_{u,v} + (l^k_v - l^k_u), \quad \forall e_{u,v} \in E' \tag{10}$$

where $w^k_{u,v}$ represents edge weight in the k-th iteration, and $(l^k_v - l^k_u)$ is the skew increment of the k-th iteration.

2) Iterative Sequential Graph Update Process

Fig. 3 shows a snapshot from the iterative sequential graph construction process. In each iteration, we construct a virtual endpoint for specific vertices, which are used in the skew calculation discussed in Section III-C2. After the update and extract process, the sequential graph construction for the next iteration can result in two scenarios: first, new essential edges are generated; second, a cycle is encountered, as shown in Fig. 3 with the example "u → v → z → u". The existence of a cycle indicates that it is impossible to eliminate the negative slack of all edges within the cycle through skew adjustments. And the negative slack improvement in the cycle C is up to $w^{avg}_C = (w_{u,v} + w_{v,z} + w_{z,u})/3$. According to Eq. (9), treating w^{avg}_C as T and rewriting $l_v = \beta(v) \cdot T - \alpha(v)$, we can calculate the clock latency for all vertices within the cycle C. After setting the clock latency values, we fix them within the cycle, update the timing propagation, and refrain from arranging clock latency to the vertices within the cycle afterward. Finally, we define the termination condition for the algorithm's iteration as the lack of new latency increments across all vertices, indicated by $w^{k+1}_{u,v} = w^k_{u,v}$ in Eq. (10).

C. Skew Calculation on Partial Sequential Graph

Based on Eq. (3), skew represents the latency difference between two flip-flops. Since skew calculation is performed on an arborescence, we set the root latency to zero as the baseline. The latencies of other flip-flops in the arborescence are calculated relative to the baseline. In this section, we will first introduce the constraints on latency, then propose a method for constructing a non-negative latency arborescence, and finally present the complete process for calculating latency.

1) Clock Latency Constraints

Similar to IC-CSS, we set all latencies to be non-negative. During the algorithm's iterative process, we gradually increase the latency of the flip-flops until the final upper bound is reached. The upper bound of latency is constrained by two factors: first, encountering the cycle illustrated in Section III-B2, indicating that further slack optimization is no longer possible. Second, latency cannot arise from another type of slack (for example, not introducing new early violations while optimizing late violations).

According to Eq. (3), in CSS, the late slack increment and early slack increment are mutually constrained. In the following discussion, we focus on late slack optimization, as the case for early slack is analogous. As illustrated in Fig. 4, when there is a late timing

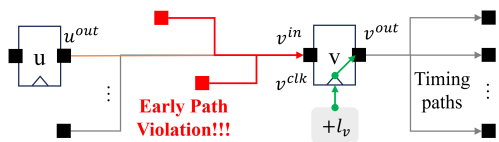

Fig. 4: Increasing clock latency for late slack improvement may introduce new early slack violations.

violation in the timing path from flip-flop u to flip-flop v, we can improve the slack by increasing the clock latency l_v at the clock input of flip-flop v. According to Eq. (1), increasing l_v may lead to new early violation paths with v^{in} as the timing path endpoint. Therefore, the increase in l_v is limited by an upper bound $\hat{s}^E_v$.

$$\hat{s}^E_v = \max\{0, s^E_v\} \tag{11}$$

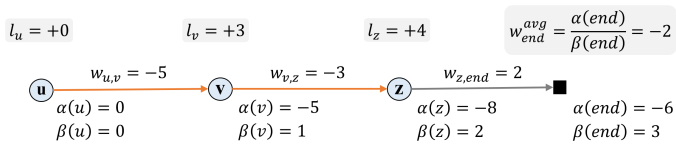

Fig. 5: Latency calculation on a non-decreasing arborescence.

where s_v^E is the minimum early slack among all timing paths with v as the endpoint. Notably, during clock skew scheduling, s_v^E varies dynamically as latency changes. Generally, during clock skew scheduling, all early paths ending at v must be extracted to ensure the upper bound $\hat{s}_v^E$ is not violated. However, our algorithm performs timing propagation in each iteration, during which the timer updates s_v^E. In other words, our algorithm update the latency upper bound $\hat{s}_v^E$ for node v without extracting its early paths, as shown in Eq. (11). This is another key factor enabling our algorithm to minimize sequential edge extraction.

2) Construction of Non-Negative Latency Arborescence

In IC-CSS, the arborescence is constructed using the parametric shortest path algorithm. Since the parametric shortest path algorithm is not the focus of this paper, we do not provide additional details here. The detailed process of arborescence generation can be referred to the Algorithm 1 in [9]. After constructing the arborescence, we calculate latencies based on its structure. We claim that if the edge weights of an arborescence A are non-decreasing from the root to the leaf nodes, the latencies of all vertices on A satisfy $l_u \geq 0$, $\forall u \in A$. Instead of a formal proof, we illustrate this with the example in Fig. 5. In this figure, the edge weights are increasing, and the calculations of α and β follow Eq. (7). The average terminal weight, w_{end}^{avg}, is first computed, after which the latency of each vertex is calculated using the formula $l_u^{max} = \beta(u) \cdot w_{end}^{avg} - \alpha(u), \quad u \in A$. For example, the latency of node z is $l_z = 2 \cdot (-2) - (-8) = +4$. The same method applies to nodes u and v. As a result, the latencies of all nodes satisfy $l_u \geq 0$, $\forall u \in A$. To ensure a non-decreasing arborescence, it suffices to enforce $w_{u,v} < w_v^{out}$ when adding edge $e_{u,v}$ to the arborescence.

3) Clock Latency Calculation by Two-pass Traversal

To ensure that the allocated clock latency for the target vertices remains within the established bounds while maximizing slack improvement, we propose a two-pass traversal method for latency calculation. In the first pass, a reverse topological traversal determines the maximum allowable latency for each vertex. In the second pass, a topological traversal computes the actual latency values.

For each vertex u, we introduce w_u^{avg}, which considers the maximum allowable latency of its successor vertices:

$$w_u^{avg} = \max_{\{u,v\} \in E} \{ \frac{\alpha(u) + w_{u,v} + l_v^{max}}{\beta(u) + 1} \} \tag{12}$$

Using w_u^{avg}, the maximum allowable latency l_u^{max} is calculated as:

$$l_u^{max} = \beta(u) \cdot w_u^{avg} - \alpha(u) \tag{13}$$

Additionally, l_u^{max} can be utilized to enforce the latency upper bound

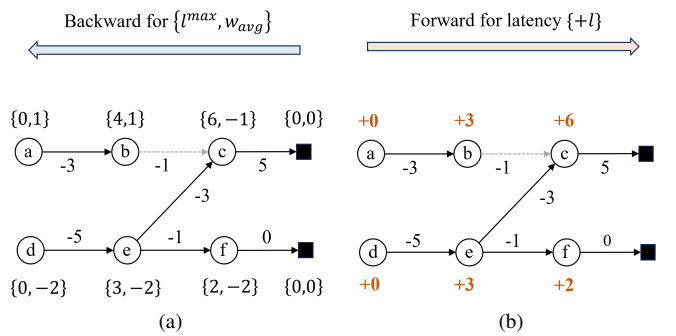

Fig. 6: An example of two-pass traversal for latency calculation.

Algorithm 1 Our Iterative Clock Skew Scheduling

Input: The gate-level timing graph G_t; The sequential graph $G = (V, E, w)$, E and w is given implictly by G_t.
Output: The target latency l_v^* for each node $v \in V$

1: Initialize Target Latency $l_v^* = 0$ $(\forall v \in V)$, Arborescences A, $k = 0$
2: **repeat**
3: $G^k \leftarrow (V, \text{extract\_edges}(G_t), w)$ [Section III-B1]
4: $A \leftarrow$ Non-Negative Construction [Section III-C2]
5: **if** exist cycle C in A **then**
6: $l_v^k \leftarrow$ Cycle Latency Calculation [Section III-B2]
7: $l_v^* = l_v^* + l_v^k$
8: Update Timing & Continue
9: **end if**
10: $l_v^k \in V \leftarrow$ Two-Pass Calculation [Section III-C3]
11: $l_v^* = l_v^* + l_v^k$
12: Update Timing & $(k = k + 1)$
13: **until** $l_v^k = 0$, $\forall v \in V$
14: **return** l_v^*

constraints outlined in Section III-C1. Finally, the actual latency l_v is scheduled in topological order:

$$l_v = \min\{l_v^{max}, l_u - w_{u,v}\} \tag{14}$$

To illustrate the latency calculation process, we refer to Fig. 6. In Fig. 6(a), the traversal begins from a virtual end node. The l_{end}^{max} for all virtual vertices is set to 0. Since virtual vertices lack outgoing edges, w_{end}^{avg} is therefore 0. The traversal continues, calculating up to vertex e using Eq. (12) and Eq. (13). For example, when calculating from vertex e to c, we find: $w_e^{avg} = \frac{(-5)+(-3)+6}{1+1} = -1$. Similarly, for the edge from e to f: $w_e^{avg} = \frac{(-5)+(-1)+2}{1+1} = -2$. The value $w_e^{avg} = -2$ is then used to calculate l_e^{max}. For the edge $e_{b,c}$, even though vertices b and c belong to different arborescences, the calculation method remains consistent. Ultimately, all $\{l^{max}, w_{avg}\}$ pairs are determined for latency calculation according to Eq. (14), as shown in Fig. 6(b). Notably, vertex b requires only a latency of +3 to resolve the timing violation on edge $e_{a,b}$.

D. Algorithm and Analysis

Alg. 1 presents the complete process of iterative clock skew scheduling. After initializing the necessary variables (Line 1), the algorithm extracts essential edges to construct the sequential graph G^0 for the first iteration (Line 3). Note that we use the function extract_edges(G_t) to invoke the timer for essential edge extraction. It then constructs the non-negative latency arborescences for skew calculation (Line 4). When a cycle C occurs in the arborescence, the algorithm calculates and records the latencies in that cycle (Lines 5-8). It then fixes the latency on C and reconstructs the sequential graph. Otherwise, it performs the two-pass latency calculation (Line 10). The algorithm records each iteration's target latency l^* (Lines 7,11). After finishing the clock skew scheduling, it puts the latency to each flip-flop and performs timing propagation for the next iteration (Lines 8,12). Finally, if all latencies are equal to zero, indicating no further slack improvement, the algorithm records k and returns all the target latencies.

IC-CSS extracts the outgoing edges for the critical vertices, potentially giving a time complexity up to $O(nm')$, where n is the number of flip-flops, and m' is the number of nets in the gate-level timing graph. Our algorithm performs timing propagation at each iteration. Therefore, the final time complexity for our sequential graph extraction is $O(km')$, where k is the number of iterations. Generally, k is much smaller than n. The significant reduction in time for our sequential graph extraction is primarily due to two factors: first, the absence of MMWC identification; and second, our focus on searching only essential edges during iterations.

E. The Modified Incremental Clock Skew Scheduling

We adapt and extend the IC-CSS algorithm from [9] to address the same NSO problem as our proposed algorithm, with modifications to satisfy latency constraints in clock skew scheduling. The following three functionalities are modified:

i. Termination Condition Adjustment: The termination condition of the IC-CSS algorithm is replaced with the cycle latency calculation method described in Section III-B2. The algorithm continues generating arborescences after processing the cycle.
ii. Latency Constraint Edge Extraction: We introduce functionality to extract latency constraint edges, as outlined in the extension of [9]. When the IC-CSS algorithm adds a critical edge $e_{u,v}$ to the arborescence, it computes the latency l_v for vertex v and compares it with $\hat{s}_v^E$ (or $\hat{s}_v^L$ for early improvement), as defined in Section III-C1. If $l_v > \hat{s}_v^E$, it triggers a callback mechanism to extract all constraint edges associated with vertex v.
iii. Latency Calculation Update: The latency calculation in IC-CSS is replaced with the approach described in Section III-C3. For each arborescence generated by IC-CSS, latencies are computed, and edge weights are updated according to Eq. (10). Subsequently, the $\hat{s}_v^E$ values for all vertices are refreshed. This process iterates until no further latency increments are possible.

We refer to the modified version of the IC-CSS algorithm as IC-CSS+.

IV. Slack Optimization Techniques

In this section, we introduce two slack optimization techniques: LCB-FF reconnection and cell movement.

A. LCB-FF Reconnection

As shown in Fig. 7(a), we introduce a flip-flop reconnection technique to local clock buffers (LCBs) to effectively increase clock latency. Alg. 1 provides the target latency l^* for flip-flops. Accordingly, the objective of LCB-FF reconnection can be formulated as:

$$\min |l_v - l_v^*| \tag{15}$$

where l_v represents the actual latency of vertex v. However, we observed that when an LCB is connected to multiple flip-flops located far apart, it can result in uncontrollable Clock Pessimism Path Reduction (CPPR) issues. We should mitigate the negative impact on the steiner tree topology caused by reconnecting an LCB to multiple flip-flops. Therefore, we prohibit reconnection for two types of LCBs: first, when the LCB's fanout has reached its maximum limit, and second, when the LCB has already undergone reconnection. We start flip-flop reconnection in descending order of l_v^* values, using the Elmore delay model to convert l_v^* into target distance $Dist_v^*$.

$$Dist_v^* = Elmore(l_v^*) \tag{16}$$

We then define the number of candidates and select LCBs for the candidate set from the distance matrix in descending order based on $Dist_v^*$. Notice that we maintain a distance matrix between the flip-flops that require reconnection and all LCBs during initialization. Finally, the reconnection cost is computed based on Eq. (15). In practice, we incorporate the negative impact on other flip-flops into the reconnection cost.

B. Cell Movement to Refine Early Violations

As shown in Fig. 7(b), we employ cell movement refinement to mitigate newly generated early violations. We first identify all movable cells along the early violation path and attempt to shift each cell in the north, south, east, and west directions. After each move, we perform a local timing update. If a cell achieves a longer arrival time for early slack improvement, further movement of that cell is halted. The cell movement distance is centered around the current position, starting at 0.1 times the maximum displacement constraint and gradually increasing until the maximum displacement is reached.

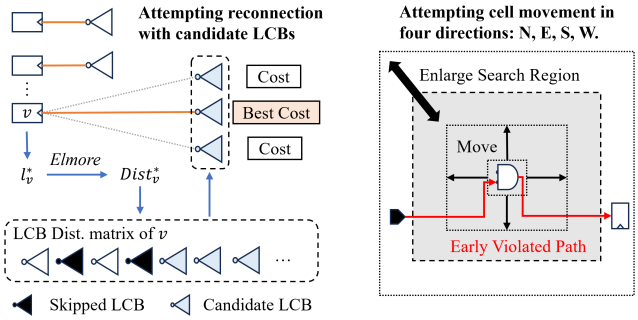

Fig. 7: The proposed slack optimization techniques: (a) LCB-FF reconnection and (b) Cell movement for early violations.

V. Experimental Results

Our CSS algorithm and slack optimization techniques are implemented in C++11 and executed on an Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz with 64GB of RAM. Timing propagation and essential edge extraction are conducted using the open-source timer described in [13]. The evaluation dataset is derived from the ICCAD 2015 contest benchmark [19], with detailed information summarized in TABLE I. To demonstrate the effectiveness of clock skew scheduling for slack improvement, we utilize the first-place result of the ICCAD 2015 contest as the input for our incremental timing optimization. Notably, we adhere to the constraint that each LCB can connect to a maximum of 50 flip-flops. All results are evaluated using the official ICCAD 2015 evaluator, and no violations of the contest constraints are observed in our experimental results. We divide our slack optimization process into two stages: early slack optimization under late slack constraints, followed by late slack optimization under early slack constraints.

TABLE I presents the comprehensive early/late worst negative slack (WNS) and total negative slack (TNS), runtime details including clock skew scheduling (CSS), slack optimization (OPT), the number of extracted edges, and the increase in HPWL. For early optimization alone, our algorithm (Ours-Early) achieves a 22.74% further improvement in WNS and a 7.22% in TNS compared to the Fast Predictive Useful Skew Methodology (FPM), with a 27.01× speedup over FPM, accompanied by a negligible increase in HPWL.

In the complete slack optimization task, our algorithm and the Modified Incremental Clock Skew Scheduling algorithm (IC-CSS+) achieve an 87.5% improvement in early WNS and approximately 88% improvement in early TNS compared to the baseline (Contest 1st). For late optimization, both achieve a 2.02% improvement in WNS and approximately 12.3% improvement in TNS. During the clock skew scheduling phase, our proposed iterative clock skew scheduling method is 49.11× faster, as it reduces the extraction of essential edges by 90.05%. Even after applying the same slack optimization phase, our proposed slack improvement method still achieves an 11.83× speedup. Experimental results demonstrate that clock skew scheduling can be accomplished with significantly less sequential graph information, enabling a rapid evaluation of the potential of clock skew for negative slack optimization.

According to TABLE I, our algorithm eliminates all early violations except for superblue7, which demonstrates its effectiveness. Furthermore, we report runtime information for Ours-Early. The results indicate that CSS is completed within a few seconds, since only a small number of edges are extracted. The runtime bottleneck is primarily due to the OPT phase. These findings highlight the efficiency of our algorithm in clock skew scheduling and underscore the potential of useful skew for optimizing negative slack.

As shown in Fig. 8, we present the iterative process of our algorithm on superblue18. The algorithm first performs early clock skew

TABLE I: Comparison of the Fast Predictive Useful Skew Methodology (FPM) [3], the Modify Incremental Clock Skew Scheduling (IC-CSS+) based on [9], and our algorithm. Since FPM only optimizes for the early violations, we introduce Ours-Early for a fair comparison. The runtime of FPM is reconstructed based on Figure 7(b) in [3]. Similar to our algorithm, IC-CSS+ conducts comprehensive optimization for both early and late violations. All algorithms use the Contest 1st results as input.

Benchmark	Statistics			Solution	Early(ps)		Late(ns)		Runtime(s)			#Extract Edge	HPWL Incr(%)
	#Cells	#FFs	#LCBs		WNS	TNS	WNS	TNS	CSS	OPT	Total		
superblue1	1.21M	144K	7.2K	Contest 1st	-16.65	-80.89	-4.57	-351.23	—	—	—	—	—
				FPM [3]	-0.43	-0.43	-4.57	-351.21	—	—	495	—	0.0003
				Ours-Early	0	0	-4.57	-351.23	1.75	22.36	24.11	32	0.0016
				IC-CSS+	0	0	-4.51	-316.94	1335.73	114.38	1450.11	3212965	0.1619
				Ours	0	0	-4.51	-316.95	7.47	97.52	104.99	81002	0.1576
superblue3	1.21M	168K	8.4K	Contest 1st	-13.13	-214.03	-8.71	-1160.04	—	—	—	—	—
				FPM [3]	-5.54	-29.05	-8.70	-1160.07	—	—	1330	—	0.0046
				Ours-Early	0	0	-8.71	-1160.20	2.18	21.52	23.7	77	0.0101
				IC-CSS+	0	0	-8.44	-1107.61	249.11	118.59	367.7	384685	0.2626
				Ours	0	0	-8.44	-1107.63	7.05	106.32	113.37	21769	0.2635
superblue4	796K	177K	8.8K	Contest 1st	-12.28	-53.84	-5.76	-2464.56	—	—	—	—	—
				FPM [3]	0	0	-5.76	-2462.93	—	—	358	—	0.0006
				Ours-Early	0	0	-5.76	-2464.53	2.39	12.99	15.38	47	0.0017
				IC-CSS+	0	0	-5.76	-2222.36	2512.37	158.22	2670.59	4524898	1.7098
				Ours	0	0	-5.76	-2223.34	92.68	125.55	218.23	802408	1.6994
superblue5	1.09M	114K	5.7K	Contest 1st	-36.77	-618.27	-24.29	-5842.23	—	—	—	—	—
				FPM [3]	-36.77	-268.60	-24.29	-5842.28	—	—	755	—	0.0038
				Ours-Early	0	0	-24.29	-5842.23	2.85	23.93	26.78	177	0.0096
				IC-CSS+	0	0	-24.29	-5065.66	2645.23	220.94	2866.17	8918052	1.7802
				Ours	0	0	-24.29	-5066.23	20.31	240.92	261.23	30380	1.7831
superblue7	1.93M	270K	13.5K	Contest 1st	-6.75	-1958.34	-15.22	-1510.76	—	—	—	—	—
				FPM [3]	-6.38	-1858.48	-15.21	-1510.79	—	—	1237	—	0.0008
				Ours-Early	-6.75	-1872.18	-15.22	-1510.69	3.35	61.91	65.26	449	0.0023
				IC-CSS+	-6.75	-1870.91	-15.22	-1387.73	803.96	232.11	1036.07	502292	0.5069
				Ours	-6.75	-1874.54	-15.22	-1388.43	31.73	227.50	259.23	5610	0.5084
superblue10	1.88M	241K	12.1K	Contest 1st	-5.15	-373.75	-16.08	-31517.8	—	—	—	—	—
				FPM [3]	-2.20	-3.47	-16.07	-31518.0	—	—	1049	—	0.0010
				Ours-Early	0	0	-16.08	-31518.1	3.19	37.10	40.29	220	0.0060
				IC-CSS+	0	0	-15.83	-29449.7	11006.17	411.93	11418.1	15481826	3.5442
				Ours	0	0	-15.83	-29448.8	202.55	378.89	581.44	2395607	3.5442
superblue16	982K	143K	7.1K	Contest 1st	-7.55	-37.64	-3.85	-265.57	—	—	—	—	—
				FPM [3]	0	0	-3.84	-265.57	—	—	430	—	0.0006
				Ours-Early	0	0	-3.85	-265.57	1.28	13.53	14.81	25	0.0005
				IC-CSS+	0	0	-3.49	-181.98	212.93	100.00	312.93	429939	0.4397
				Ours	0	0	-3.49	-182.94	7.00	89.01	96.01	4147	0.4412
superblue18	768K	104K	5.2K	Contest 1st	-1.95	-6.86	-3.82	-775.84	—	—	—	—	—
				FPM [3]	0	0	-3.81	-775.87	—	—	301	—	0.0015
				Ours-Early	0	0	-3.82	-775.84	0.91	9.24	10.15	11	0.0006
				IC-CSS+	0	0	-3.78	-659.27	182.88	72.19	255.07	347689	0.8218
				Ours	0	0	-3.78	-659.24	15.17	72.60	87.77	22508	0.8221
Avg. Ratio (Based on 1st)				FPM [3]	+64.76%	+80.83%	+0.06%	+0.01%	—	—	+744.38s	—	**-0.0017%**
				Ours-Early	**+87.50%**	**+88.05%**	+0.00%	+0.00%	+2.24s	+25.32s	**+27.56s**	+130	-0.0041%
				IC-CSS+	+87.50%	**+88.06%**	+2.02%	**+12.33%**	+2368.55s	+178.54s	+2547.09s	+4225293	-1.1534%
				Ours	+87.50%	+88.03%	+2.02%	+12.27%	**+47.99s**	**+167.29s**	**+215.28s**	**+420429**	**-1.1524%**
Avg. Improvement				FPM [3]	**+22.74%**	**+7.22%**	-0.06%	-0.01%	—	—	**27.01×**	—	-0.0024%
				IC-CSS+	+0.00%	-0.03%	+0.00%	-0.06%	**49.11×**	1.04×	**11.83×**	**-90.05%**	+0.001%

scheduling, followed by early slack optimization using the LCB-FF reconnection and cell movement techniques. Similarly, the algorithm carries out late slack optimization. To enhance stability, a portion of early violations is amplified during early slack optimization. Notably, the first round of late clock skew scheduling reveals a significant slack improvement, indicating that substantial skew is available on adjacent paths to help resolve timing violations.

VI. Conclusion

In this paper, we address the negative slack improvement problem through a fast, iterative clock skew scheduling algorithm. This method rapidly evaluates the potential for slack improvement and uses the calculated latencies to guide slack optimization techniques, thereby achieving significant timing improvements and runtime reduction. Furthermore, our algorithm supports controlling flip-flop clock latency constraints, enabling customized clock skew scheduling. Fast clock skew scheduling enables the incorporation of useful skew considerations into the EDA physical design flow. In future work, we plan to apply our algorithm to open-source flows to guide clock tree synthesis and integrate it with logic path optimization.

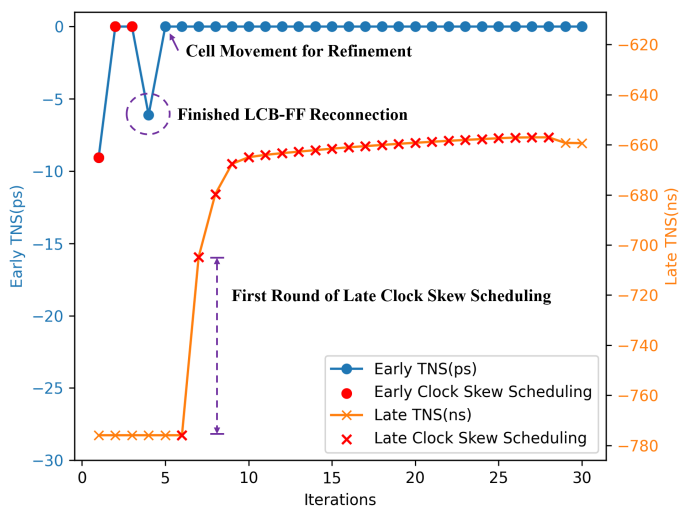

Fig. 8: Slack optimization iterations of superblue18.

REFERENCES

[1] Y.-C. Lu, W.-T. Chan, D. Guo, S. Kundu, V. Khandelwal, and S. K. Lim, "RL-CCD: Concurrent clock and data optimization using attention-based self-supervised reinforcement learning," in *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[2] T.-B. Chan, A. B. Kahng, and J. Li, "NOLO: A no-loop, predictive useful skew methodology for improved timing in IC implementation," in *Proceedings of the 15th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2014, pp. 504–509.

[3] S. Kim, S. Do, and S. Kang, "Fast predictive useful skew methodology for timing-driven placement optimization," in *Proceedings of the 54th ACM/IEEE Design Automation Conference (DAC)*, 2017, pp. 1–6.

[4] J. P. Fishburn, "Clock skew optimization," *IEEE transactions on computers*, vol. 39, no. 7, pp. 945–951, 1990.

[5] M. Ni and S. O. Memik, "A revisit to the primal-dual based clock skew scheduling algorithm," in *Proceedings of the 11th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2010, pp. 755–764.

[6] G. Zgheib, Y. S. Lu, and I. Ganusov, "Clock skew scheduling: Avoiding the runtime cost of mixed-integer linear programming," in *Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 327–333.

[7] A. Wagle, J. Yang, N. Kulkarni, and S. Vrudhula, "A new approach to clock skewing for area and power optimization of ASICs using differential flipflops and local clocking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 4164–4176, 2023.

[8] C. Albrecht, B. Korte, J. Schietke, and J. Vygen, "Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip," *Discrete Applied Mathematics*, vol. 123, no. 1-3, pp. 103–127, 2002.

[9] C. Albrecht, "Efficient Incremental Clock Latency Scheduling for Large Circuits," in *Proceedings of the Design Automation & Test in Europe Conference (DATE)*. Munich, Germany: IEEE, 2006, pp. 1–6.

[10] K. Wang, H. Fang, H. Xu, and X. Cheng, "A fast incremental clock skew scheduling algorithm for slack optimization," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference (ASPDAC)*. IEEE, 2008, pp. 492–497.

[11] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 895–902.

[12] "OpenSTA", *https://github.com/The-OpenROAD-Project/OpenSTA*.

[13] G. Flach, M. Fogaça, J. Monteiro, M. Johann, and R. Reis, "Rsyn: An extensible physical synthesis framework," in *Proceedings of the 2017 ACM on International Symposium on Physical Design (ISPD)*, 2017, pp. 33–40.

[14] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. Wong, "OpenTimer v2: A new parallel incremental timing analysis engine," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 40, no. 4, pp. 776–789, 2020.

[15] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated critical path generation with path constraints," in *Proceedings of the 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[16] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating static timing analysis using CPU–GPU heterogeneous parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 4973–4984, 2023.

[17] T.-W. Huang, B. Zhang, D.-L. Lin, and C.-H. Chiu, "Parallel and heterogeneous timing analysis: Partition, algorithm, and system," in *Proceedings of the 2024 International Symposium on Physical Design (ISPD)*, 2024, pp. 51–59.

[18] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "GCS-Timer: Gpu-accelerated current source model based static timing analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024, pp. 1–6.

[19] M.-C. Kim, J. Hu, J. Li, and N. Viswanathan, "ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite," in *Proceedings of the 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2015, pp. 921–926.