Optimal Wire Retiming Without Binary Search*

Chuan Lin and Hai Zhou
Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208

Abstract

The problem of retiming over a netlist of macro-blocks to achieve the minimal clock period, where the block internal structures may not be changed and flip-flops may not be inserted on some wire segments, is called the optimal wire retiming problem. To the best of our knowledge, there is no polynomial-time approach to solve it and the existence of such an approach is still an open question. In this paper, we present a brand new algorithm that solves the optimal wire retiming problem with polynomial-time worst case complexity. Since the new algorithm avoids binary search and is essentially incremental, it has the potential of being combined with other optimization techniques. Experimental results show that the new algorithm is very efficient in practice.

1 Introduction

With a great market drive for high performance and integration, operating frequencies and chip sizes of System-On-Chip (SOC) are dramatically increasing. Industry data showed that the frequencies of high-performance ICs approximately doubled every process generation and the die size also increased by about 25% per generation. With such short clock periods, the communication among different blocks on an SOC circuit of ever increasing complexity is becoming a bottleneck: even with interconnect optimization techniques such as buffer insertion, the delay from one block to another may still be longer than one clock period, and multiple clock cycles are generally required to communicate such a global signal.

This trend has motivated recent research within Intel [3] and IBM [8] on how to insert flip-flops on a given net if the communication between the pins requires multiple clock cycles. However, inserting flip-flops within a circuit will change its functionality, and inserting arbitrary number of them on a net without considering global consistency will destroy the correctness of a circuit.

Retiming [12] is a traditional sequential optimization technique that moves flip-flops within a circuit without destroying its functionality. In traditional settings, retiming was used mainly on gate level netlists [16, 7, 11, 15, 19]. Although some research incorporated wire delays in retiming [11, 17, 5], they did not consider the situation where multiple flip-flops may be on a global interconnect. Not until recently [13, 2, 14, 18], the alternative utility of retiming–that is, besides its computational function, a flip-flop can be used to fulfill communication buffering requirements–has been explored.

Since dominant wire delays can only happen on global wires, it is more meaningful to formulate the problem at the chip level as in [13, 14], that is, the design we deal with is a netlist of macro-blocks. The wires within a block are relatively much shorter thus do not need multiple clock periods for propagation. In SOC design, many of the macro-blocks are IP (Intellectual Property) cores. Some of these blocks may be combinational circuits, and others sequential. Because of the existence of pre-designed blocks such as IP cores or regular-structured blocks such as memories, (combinational) buffers or flip-flops may not be inserted everywhere [20].

In this application, the approaches in [2] cannot be used because they took into account only gates and cannot be extended to handle complex blocks. On the other hand, our previous work [13] solved the problem with complex blocks by proposing timing macro-models to model the timing behavior of the blocks, based on which a set of integer difference inequalities was shown to be both necessary and sufficient, thus quantify a feasible solution. Although it gave a polynomial-time algorithm for feasibility checking, the complexity was high, making it inhibitive even for checking circuits with about 1000 vertices. Furthermore, it only gave a fully polynomial-time approximation scheme (FPTAS) for clock period minimization, that is, the overall complexity was dependent on a given precision. The same thing was also true in [2] and our improved work [14].

Other FPTASs were proposed in [9], where the binary search approach was used to deal with retiming in the continuous domain. Although a tight lower and upper bound can be obtained, the candidates of possibly feasible clock periods are innumerable in the continuous domain, which is different from Leiserson and Saxe [12]. Therefore, a precision is required to terminate the binary search. However, the introduction of precision leads to a trade-off between the accuracy and the computational complexity. Theoretically, it fails to be a polynomial-time approach as no precision is sufficient for exact solution.

In this paper, a polynomial-time incremental algorithm is proposed to compute the minimal clock period without using the binary search. To the best of our knowledge, it is the first work that shows the optimal wire retiming problem in the continuous domain can be solved in polynomial time.

2 Problem formulation

The problem we want to solve is the same as [13, 14], which is the retiming on an SOC design with a given block placement (also known as floorplan) and a global routing of the global wires. In Figure 1(a), we show a typical SOC design, including a combinational block, a sequential block, and a net part of which goes through buffer forbidden areas (illustrated as shaded regions). Different blocks can also be classified into two categories: complete bipartite and non-complete bipartite. For the example in Figure 1(a), the sequential block is complete bipartite, while the combinational block is non-complete bipartite as there is no path from b to p. We will also use the same timing models to specify the timing behavior of the circuit. To avoid repetition, we only outline the models below. Interested readers may refer to [13] for details.

The model for a combinational block is straightforward. Edges between inputs and outputs are introduced to represent the path delays between them. Since we only care about the set-up conditions of flip-flops, if minimum and maximum delay pairs are given for a combinational block, only the maximum delays are taken. In the model for a sequential block, a

*This work was supported by the NSF under CCR-0238484.

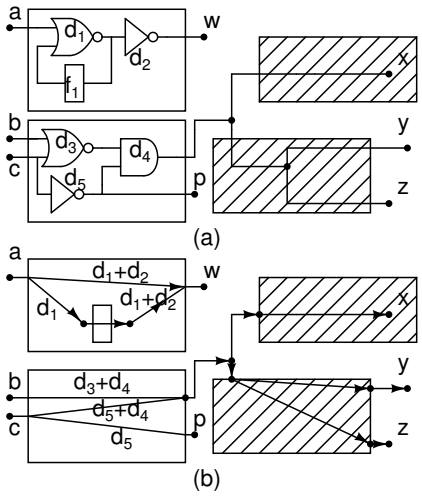

Figure 1: A circuit example and its timing model

virtual flip-flop is introduced to enforce the set-up conditions of the block inputs whenever there is a path from an input to an interior flip-flop. It is also used to specify the arrival times of the block outputs whenever a path exists from an interior flip-flop to an output. Later, the virtual flip-flop is further modeled as an edge with delay zero and weight one. Modeling a net consists of two parts: the part within buffer forbidden areas is treated as a combinational block; the other as a Steiner tree. On those wire segments where buffers and flip-flops are allowed, linear delays are assumed based on optimal buffering. For others in buffer forbidden areas, delays are computed using the Elmore delay model with the assumption that the boundary of buffer forbidden areas is all buffered.

In short, with the macro-models, an SOC design becomes a directed graph, where a vertex represents a source or sink of a net, the input or output of the virtual flip-flop introduced by a sequential block, a point where a wire gets into or out of a buffer forbidden area, or a Steiner point outside buffer forbidden areas. A directed edge represents a fan-out relation within a block or a wire connection outside buffer forbidden areas. On an edge representing a fan-out relation within a block, the delay is a nonnegative constant and no flip-flop change can be made on it. It is also called a forbidden edge. On the other hand, flip-flops can be moved into or out of an edge representing a wire outside buffer forbidden areas. The delay of such a wire is positive and proportional to its length. After applying these macro-models, the timing model for the design in Figure 1(a) is illustrated in Figure 1(b). The problem we want to solve can be formulated as follows.

Problem 1 (Optimal Wire Retiming)
Given a directed graph $G = (V, E)$ with two types of edges $E = E_1 \cup E_2$, where each edge $e \in E$ has a delay $d(e)$ and a weight (representing number of flip-flops) $w(e)$, find a retiming–i.e. a relocation of flip-flops in the graph–such that: 1. there is no flip-flop change on any edge $e \in E_1$; 2. the delay between two flip-flops on an edge $e \in E_2$ is linear in terms of their distance; 3. the clock period (i.e. the maximum delay between any two consecutive flip-flops, treating primary inputs/outputs as flip-flops) is minimized.

3 Notations and constraints

From the formulation of the problem, we already have a delay label $d : E \to R^+$ and a weight label $w : E \to Z^+$. We will adopt the tradition to use a label $r : V \to Z$ to denote the number of flip-flops moved over a vertex and a label $t : V \to R$ to denote the arrival time of the vertex. Similarly, the arrival time of a flip-flop is the arrival time of its input.

For any path $p \in G$, we use $d(p)$ to denote the delay along p, which is the sum of the delays of p's constituent edges. Similarly, $w(p)$ denotes the number of flip-flops on p before retiming, which is the sum of the weights of p's constituent edges. To ease the representation, we use $w_r(u, v)$ to denote the number of flip-flops on $(u, v) \in E$ after retiming, i.e.,

$$w_r(u, v) = w(u, v) + r(v) - r(u).$$

We also use $w_r(p)$ to denote the number of flip-flops on $p = u \rightsquigarrow v$ after retiming: $w_r(p) = w(p) + r(v) - r(u)$. When a path actually forms a cycle c, $d(c)$ ($w(c)$) includes the delay (weight) of each edge in the cycle only once. Since retiming will not change the number of flip-flops in a cycle, $w(c)$ is independent of retiming. To avoid disturbing issues regarding zero weight cycles, we assume in this paper that $w(c) > 0$ for all $c \in G$. The cycle ratio of c is defined as $\rho(c) = d(c)/w(c)$ [6].

Similar to [13], a vertex M is introduced into G, as well as directed forbidden edges from each primary output to it with zero delays and weights, and from it to each primary input with delay zero but weight one.

3.1 Constraints for retiming validity

A valid retiming must satisfy the following constraints.

$$r(u) = r(v), \qquad \forall (u, v) \in E_1 \tag{1}$$
$$w_r(u, v) \geq 0, \qquad \forall (u, v) \in E_2 \tag{2}$$

3.2 Constraints for timing validity

Let T denote a given clock period. Timing validity refers to the following constraints.

$$t(v) = \max\Big(0, \max_{\forall (u,v) \in E} t(u) + d(u, v) - w_r(u, v)T\Big), \forall v \in V \tag{3}$$
$$t(v) \leq T, \quad \forall v \in V \tag{4}$$

Edge (u, v) is called a *critical edge* if $t(v) = t(u) + d(u, v) - w_r(u, v)T$. Likewise, *critical paths* and *critical cycles* can be similarly defined. Our task is to find the minimal T with which both retiming and timing validities are satisfied.

4 Algorithm

Our strategy is to first construct a starting point, i.e., a retiming satisfying (1)-(4), and then gradually target the minimal clock period with (1)-(4) inviolate.

4.1 Initialization

Naturally, the original flip-flop configuration satisfies retiming validities, and can be initialized to satisfy timing validities as well. A simple procedure as in Figure 2 will generate a starting point satisfying (1)-(4) with $r(v) = 0, \forall v \in V$. The idea is to line up the flip-flops on each edge $(u, v) \in E$ together immediately before v. When completed, it returns a clock period T and the corresponding t values.

4.2 Base algorithm

Suppose there exists another retiming $(\bar{r}, \bar{t}, \bar{T})$ satisfying (1)-(4) with $\bar{T} < T$. Intuitively, to reduce T to $\bar{T}$, there are two possible adjustments. One is to locally reposition flip-flops under r, that is, without changing the number of flip-flops on each edge. The other allows such changes. A straightforward strategy is to iteratively apply these two adjustments in a way that is guaranteed to lead to the minimal clock period. We

```
INIT(G, r)
  w_r(u,v) ← w(u,v) + r(v) − r(u), ∀(u,v) ∈ E;
  t(v) ← 0, ∀v ∈ V; T ← 0; Q ← V;
  While (Q ≠ ∅) do
    u ← dequeue(Q);
    For each (u,v) ∈ E do
      If ((w_r(u,v) = 0) ∧ (t(v) < t(u) + d(u,v))) then
          t(v) ← t(u) + d(u,v);
          Q ← Q ∪ {v} if v ∉ Q;
      If (T < t(u) + d(u,v)) then
          T ← t(u) + d(u,v);
```

Figure 2: Pseudocode of initialization procedure

will outline our algorithm and explain how to reduce T under r in this section, and elaborate the second adjustment in the next section.

To reduce T under r, we actually want to find the minimal T under r with which (3)-(4) are satisfied. If we neglect (4) for a moment, it reveals that solving (3) is equivalent to computing the maximum cycle ratio (MCR) of G. Burns' algorithm [1], which is known to be one of the fastest algorithms for MCR computation within polynomial time [6], actually solves a general form of (3) to obtain the minimal T.

Further taking (4) into account, we found that Burns' algorithm can be extended to obtain the minimal T with which both (3) and (4) are satisfied. We describe it in Figure 3. The basic idea is to iteratively reduce T and update the corresponding arrival times t until either the minimal T under the current r is reached, or the optimality of T for the wire retiming problem is certified.

After the starting point (line 1), the algorithm enters the outer while-loop whose condition is always true. Inside it, the algorithm first identifies critical edges E_c (line 3). If E_c contains a cycle, then the algorithm terminates with the current T and r as the optimum (lines 4-5, the argument is given later in Corollary 2.1). Otherwise, G_c is a DAG, and a safe reduction θ for T is calculated (lines 12-15), which is based on the computation of the maximum number of flip-flops $\Delta(v)$ on paths from roots in G_c to v, $\forall v \in V$ (lines 7-11). The value of θ is further confined (lines 16-17). When it is finished, if $\theta > 0$, then at the end of the iteration we update t and T accordingly (lines 24-27, we will show later in Lemma 1 that the resultant new t and T will keep (3)-(4) inviolate), and record the best solution found so far in T^{opt} and r^{opt} (lines 28-29). On the other hand, if $\theta = 0$, we must have found a vertex v such that $t(v) = T$ resulting in $\theta = 0$ (line 17). We then increase $r(v)$ by 1 (the argument is given later in Lemma 3), adjust r if (1) or (2) is violated, and initialize the circuit under the new r (line 18-23). After the r adjustment, some criteria checking (elaborated later in Theorem 2 and 3) is applied to see if the optimal clock period has been obtained (line 21). If yes, we terminate the algorithm with T^{opt} and r^{opt} as the output. Otherwise, we loop back to continue a new iteration of the outer while-loop.

In Figure 4, we give the pseudocode for r adjustment.

4.3 Proof of correctness and computational complexity

To prove the correctness of the algorithm, we need to show that the algorithm always terminates and reports a retiming satisfying (1)-(4) with the minimal clock period.

First of all, we have the following lemma.

```
Algorithm  Incremental Optimal Wire Retiming
Input:  A graph representation G = (V, E).
Output:  A retiming with minimal clock period.

1  INIT(G, 0); T^opt ← T; r^opt(u) ← 0, ∀u ∈ V;
2  While (true) do
     ▷Identify critical edges in E
3    E_c ← {(u,v) ∈ E | t(v) = t(u) + d(u,v) − w_r(u,v)T};
4    If E_c contains a cycle then
5      Report T, r and exit;
6    Topological sort G_c = (V, E_c);
     ▷Compute max FF number on paths from roots in G_c
7    For v ∈ V in topological sort order of G_c do
8      If v is a root in G_c then
9        Δ(v) ← 0;
10     Else for each (u,v) ∈ E_c do
11       Δ(v) ← max{Δ(v), Δ(u) + w_r(u,v)};
     ▷Compute a safe reduction θ for T
12   θ ← ∞
13   For each (u,v) ∈ E do
14     If (Δ(u) + w_r(u,v) > Δ(v)) then
15       θ ← min{θ, (t(v) − t(u) − d(u,v) + w_r(u,v)T) / (Δ(u) + w_r(u,v) − Δ(v))};
16   For each v ∈ V do
17     θ ← min{θ, (T − t(v)) / (Δ(v) + 1)};
18     If (θ = 0) then
19       r(v) ← r(v) + 1;
20       ADJUST(G, v);
21       If Theorem 2 or 3 is applicable then
22         Report T^opt, r^opt and exit;
23       INIT(G, r);
     ▷Update t and T
24   If (θ > 0) then
25     T ← T − θ;
26     For each v ∈ V do
27       t(v) ← t(v) + θ · Δ(v);
28     If (T < T^opt) then
29       Update T^opt and r^opt with T and r;
```

Figure 3: Pseudocode of retiming algorithm

Lemma 1 *(1)-(4) are kept every time the algorithm in Figure 3 starts a new iteration of the outer while-loop.*

Proof: First of all, (1)-(4) are satisfied after initialization, i.e., before the first iteration of the outer while-loop. Inside the iteration, the algorithm either changes r (lines 18-19) or successfully reduces T (lines 24-25). For the former, an r adjustment and an initialization under the new r are immediately followed, which will keep (1)-(4) for the next iteration. For the latter, we will show that they are kept too.

When T is successfully reduced, t is also updated (lines 26-27), denoted as T' and t' respectively. Since (1)-(2) will not be violated with r unchanged, we only need to consider the effect of the update on (3)-(4). It is easy to show that (4) is kept after the update, otherwise there exists a $v \in V$ such that

$$\begin{aligned} t'(v) > T' \quad &\Rightarrow \quad t(v) + \theta \cdot \Delta(v) > T - \theta \\ &\Rightarrow \quad \theta > (T - t(v))/(\Delta(v) + 1) \end{aligned}$$

which contradicts the computation of θ in lines 16-17. To see how (3) is kept, we study critical edges E_c and non-critical edges $E - E_c$ respectively. For all $(u,v) \in E_c$, we have $\Delta(v) \geq$

```
ADJUST(G, v)
  Q ← {v};
  While (Q ≠ ∅) do
    u ←dequeue(Q);
    For each e (= (x, u) or (u, x)) ∈ E do
      If (((e ∈ E1) ∧ (r(x) ≠ r(u))) || (w_r(e) < 0)) then
        r(x) ← r(x) + 1;
        Q ← Q ∪ {x};
```

Figure 4: Pseudocode of r adjustment to truthify (1)-(2)

$\Delta(u) + w_r(u,v)$ (lines 10-11), thus after the update

$$\begin{aligned} & t'(v) - t'(u) - d(u,v) + w_r(u,v)T' \\ = \; & (t(v) + \theta \cdot \Delta(v)) - (t(u) + \theta \cdot \Delta(u)) \\ & -d(u,v) + w_r(u,v) \cdot (T - \theta) \\ = \; & (t(v) - t(u) - d(u,v) + w_r(u,v)T) \\ & +\theta(\Delta(v) - \Delta(u) - w_r(u,v)) \\ \geq \; & 0 \end{aligned}$$

For all $(u,v) \in E - E_c$, if $\Delta(v) \geq \Delta(u) + w_r(u,v)$, we have the same result as above. If $\Delta(v) < \Delta(u) + w_r(u,v)$, the result is also true otherwise

$$\theta > (t(v) - t(u) - d(u,v) + w_r(u,v)T)/(\Delta(u) + w_r(u,v) - \Delta(v)),$$

which contradicts the computation of θ in lines 14-15. Therefore, after the update, we have $t'(v) \geq t'(u) - d(u,v) + w_r(u,v)T'$, $\forall (u,v) \in E$. Then, for all vertex v that has at least one edge in E_c terminating at it, there is an edge $(u,v) \in E_c$ on which $\Delta(v) = \Delta(u) + w_r(u,v)$, and after the update, (u,v) remains critical, that is, (3) is satisfied on v. For all vertex v that has no edge in E_c terminating at it, we know $t(v) = 0$ by (3). Since v is actually a root in G_c, $\Delta(v) = 0$, thus $t'(v) = 0$. Given $t'(v) \geq t'(u) - d(u,v) + w_r(u,v)T'$, $\forall (u,v) \in E$, we know that (3) is also satisfied on v after the update.

Therefore, (1)-(4) are kept when the iteration is completed. By induction, the lemma is true. □

Examine the two places (lines 13-15 and 16-17) where θ is computed. Since $\Delta(v) \geq \Delta(u) + w_r(u,v)$ is true on all critical edge (u,v), if the final value of θ is actually determined in lines 13-15 on some edge (u,v), the edge cannot be critical, i.e., $t(v) - t(u) - d(u,v) + w_r(u,v)T > 0$, thus $\theta > 0$. On the other hand, if θ is determined in lines 16-17 on vertex v, then either $\theta = 0$ (i.e., $t(v) = T$), or $t'(v) = T'$ after the update (lines 25-27) and θ will be assigned zero in the next iteration.

Assume $\theta > 0$ all the time, the iterations will keep reducing T, and the algorithm has a provable complexity of $O(|V|^2|E|)$ [1] before a critical cycle is emerged. In fact, the occurrence of a critical cycle witnesses an optimal solution, which is stated in the corollary of the following lemma [13].

Lemma 2 *A feasible clock period T must satisfy*

$$T \;\geq\; \rho^* = \max_{\text{cycle } c \in G} \rho(c).$$

Corollary 2.1 *When the algorithm in Figure 3 detects a critical cycle, current T and r is an optimal solution.*

Alternatively, if $\theta = 0$, we know $t(v) = T$ for some $v \in V$. In fact, it implies that we have reached the minimal clock period under the current r, which is specified in the following lemma.

Lemma 3 *If $t(v) = T$ for some $v \in V$, then there exists a path $p = u \rightsquigarrow v$ such that $t(u) = 0$, all flip-flops on p have arrival times T, and any retiming satisfying (1)-(4) but with a smaller clock period must have more than $w_r(p)$ number of flip-flops on p.*

Proof: Consider the edges terminating at v. They cannot all be non-critical, otherwise $t(v) = 0$ by (3), which is kept by Lemma 1. Therefore, there must exist a critical path p from some root $u \in G_c$ to v, on which all flip-flops have arrival times T. Since u is a root, all edges terminating at u have to be non-critical, that is, $t(u) = 0$.

For the seek of contradiction, we assume that there exists a retiming $(\bar{r}, \bar{t}, \bar{T})$ with $\bar{T} < T$, such that $w_{\bar{r}}(p) \leq w_r(p)$, i.e., $\bar{r}(v) - \bar{r}(u) \leq r(v) - r(u)$. Given the path delay $d(p)$ is equal to $(w_r(p) + 1)T$, we have

$$\begin{aligned} \bar{t}(v) \;&\geq\; \bar{t}(u) + d(p) - w_{\bar{r}}(p)\bar{T} \\ &>\; \bar{t}(u) + (w_r(p) + 1)T - w_r(p)T \\ &\geq\; T \end{aligned}$$

Hence, $\bar{T} \geq \bar{t}(v) \geq T$, a contradiction. That is, $w_{\bar{r}}(p) > w_r(p)$ must be true. □

Therefore, if T is not the optimum, we must have

$$\bar{r}(v) - \bar{r}(u) > r(v) - r(u).$$

Since it is not the absolute values of r but their differences that are relevant in the retiming, we can either increase $r(v)$ or decrease $r(u)$ to approach $\bar{r}$. No matter which one is chosen, the amount of change should only be 1 since we do not want to over-adjust r. Without loss of generality, we choose to increase $r(v)$ by 1 in our implementation.

Given the unimportance of the absolute values of r, we can also conclude that there must exist an optimal retiming $(\bar{t}, \bar{r}, T^*)$ satisfying (1)-(4) with the optimal T^*, such that $\bar{r}(v) \geq 0$ for all vertex $v \in V$, and $\bar{r}(\gamma) = 0$ for some vertex $\gamma \in V$. In particular, we call such an optimal retiming *an irredundant optimal retiming*. In fact, there could be a bunch of irredundant optimal retimings. Among them, we define *the least optimal retiming* (t^*, r^*, T^*) such that for all other irredundant optimal retiming $\bar{r}$, we have $r^*(v) \leq \bar{r}(v)$, $\forall v \in V$. We then prove the existence of the least optimal retiming by showing that our algorithm will reach it in polynomial time.

Theorem 1 *The algorithm in Figure 3 will reach the least optimal retiming in polynomial time.*

Proof: For all irredundant optimal retiming $\bar{r}$, we have $r(v) = 0 \leq \bar{r}(v)$, $\forall v \in V$, after initialization. Let T_r^* denote the minimal clock period under the current r. If $T_r^* = T^*$, then current r is actually r^*. Otherwise, when T_r^* is reached, we must have $\theta = 0$. According to Lemma 3, we can locate a critical path $p = x \rightsquigarrow y$ such that

$$r(y) - r(x) < \bar{r}(y) - \bar{r}(x).$$

Since $r(x) \leq \bar{r}(x)$ and $r(y) \leq \bar{r}(y)$, if $r(y) = \bar{r}(y)$, it leads to $r(x) > \bar{r}(x)$, a contradiction. Therefore, $r(y) < \bar{r}(y)$. Thus, $r(y) + 1 \leq \bar{r}(y)$, that is, by increasing $r(y)$ by 1, the new r configuration still satisfies $r(v) \leq \bar{r}(v)$, $\forall v \in V$, for all $\bar{r}$.

By induction, $r(v) \leq \bar{r}(v)$, $\forall v \in V$, for all $\bar{r}$, is always true before we reach T^*. Let N denote the sum of all $\bar{r}$ values for any particular $\bar{r}$, i.e., $N = \sum_{v \in V} \bar{r}(v)$. Given the complexity of Burns' algorithm is $O(|V|^2|E|)$, we will reach T^* in $O(|V|^2|E| \cdot (N+1))$ time. When T^* is reached, the r at that moment is r^*. □

Based on this, we have the following theorems.

Theorem 2 *For the algorithm in Figure 3, if we have $r(v) > 0$ for all vertex $v \in V$, we can terminate the algorithm.*

Proof: Since $r(v) > 0$, $\forall v \in V$, we have $r(\gamma) > r^*(\gamma) = 0$, meaning the algorithm has reached r^* and already gone beyond it. Thus, we can safely terminate the algorithm. □

Theorem 3 *For the algorithm in Figure 3, if we have a vertex $v \in V$ such that $r(v) > N_{\text{ff}}$, where $N_{\text{ff}} = \sum_{(u,v) \in E} w(u,v)$ is the total number of flip-flops in the circuit, we can terminate the algorithm.*

Proof: Let V' be the set of vertices that can be reached from v in G regardless of edge directions, including v. Suppose $u \in V'$ is such a vertex whose $r(u) = \min_{i \in V'} r(i)$. All the possible relations between u and v are shown in Figure 5.

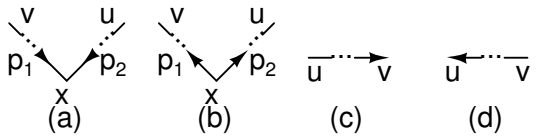

Figure 5: Four possible relations between u and v.

For case (a), we have $w_r(p_1) = w(p_1) + r(x) - r(v)$. Because there is no direct path between u and v, all paths between v and x must be from v to x. Therefore, the flip-flops that were moved into p_1 are different from those that were originally in p_1. Thus, $w(p_1) + r(x) \leq N_{\text{ff}}$. It leads to $w_r(p_1) < 0$, which however is impossible when (2) is kept by Lemma 1. For case (b), $w_r(p_1) = w(p_1) + r(v) - r(x)$. By the same argument, the flip-flops that were moved into p_1 through x must be different from those $w(p_1)$ number of flip-flops, that is, $w(p_1) + r(v) \leq N_{\text{ff}}$. Then, $r(v) \leq N_{\text{ff}} - w(p_1) \leq N_{\text{ff}}$, which contradicts $r(v) > N_{\text{ff}}$. For case (c) and (d), if $r(u) = 0$, then more than N_{ff} number of flip-flops must be moved into or out of the path between u and v, which is impossible.

Therefore, $r(u) > 0$, and there are two possible situations. If $V = V'$, it reduces to the situation in Theorem 2, by which we can safely terminate the algorithm. Otherwise, since the vertices in $V - V'$ have no connection with those in V' at all, the vertices in V' and the edges connecting them actually constitute an independent sub-circuit $G' \subset G$. The flip-flop configuration of a sub-circuit is also independent on those of others. Therefore, it must be true that $r^*(\gamma) = 0$ for some vertex $\gamma \in V'$ and $r^*(\delta) = 0$ for some vertex $\delta \in V - V'$. Given $r(\gamma) \geq r(u) > 0 = r^*(\gamma)$, we know that the algorithm has already gone beyond r^*, it then can be terminated. □

Corollary 3.1 $r^*(v) \leq N_{\text{ff}}$, $\forall v \in V$.

With Corollary 3.1 incorporated, we eventually have a polynomial bound for the computational complexity of the algorithm, which enables us to establish the correctness of the algorithm in the following theorem.

Theorem 4 *The algorithm in Figure 3 will terminate with the least optimal retiming in $O(|V|^3|E| \cdot N_{\text{ff}})$ time, where N_{ff} is the total number of flip-flops in the circuit.*

Remark 1 *The significance of the above theorem is not the actual formula of the bound, but showing that the optimal wire retiming problem can be indeed solved in polynomial time. Furthermore, caution should be used on this bound. First, a program will usually have great running time variations on different problem instances. The worst case running time may only happen on a few rare instances, thus may not be a good indication of the efficiency on most other instances. For example, the worst case assumes that the algorithm terminates when all $r(v)$, $\forall v \in V$, is increased to N_{ff}, which is hardly the case in real applications. Second, even the worst case happens on most problem instances, a bound may be loose due to the difficulty to have an accurate analysis. For example, Burns' algorithm typically performs in $O(|E|)$ time on real circuits. These assumptions lead to an ideally worst case which is very unlikely to be attainable in reality. In other words, the bound in the above theorem is very loose. Since only necessary operations are conducted in the algorithm, it should be efficient on most instances. This is confirmed by our experimental results in Section 5.*

4.4 Speed-up techniques

4.4.1 A better initialization

In practice, the majority of the running time is consumed on iteratively computing θ for T reduction. Let T_r and T_r^* denote the clock period returned by the initialization and the minimal clock period under r, respectively. Intuitively, the closer T_r is to T_r^*, the smaller the number of iterations the algorithm has to take to reach T_r^*. Recall the initialization procedure we proposed in Figure 2, we found that it gave a very loose upper bound for T_r^* because all flip-flops were simply pushed to the ending points of the edges there. Therefore, a better method for initialization that can provide a tighter bound is desired. On the other hand, we should also avoid complicated methods whose overheads might dominate the overall complexity.

To meet both requirements, we adopt the same framework as in Figure 2, while proposing an idea of *local flip-flop even distribution*. More specifically, instead of pushing all flip-flops to v for all $(u,v) \in E$ with $w_r(u,v) > 0$ during the initialization, we check if $t(u) + d(u,v) \leq (w_r(u,v)+1)T$, where T is the current maximum of t. In other words, we try to place the flip-flops on (u,v) to make their arrival times to be T, and see if the resultant $t(v)$ exceeds T. If $t(v)$ does exceed T, we then update T and $t(v)$ with $\frac{t(u)+d(u,v)}{w_r(u,v)+1}$, meaning all flip-flops on (u,v) thus have the same arrival times as $t(v)$, they are evenly distributed.

When completed, it gives a much better T_r, however, (3) might not be satisfied because now the arrival times of flip-flops on critical edges could be smaller than T_r. To truthify (3), we propose another procedure called *initialize t with a given clock period*, whose pseudocode is given in Figure 6. The basic idea is to always make the arrival time of a flip-flop to be equal to the given clock period if possible.

Applying it with T_r, the arrival times of flip-flops on critical edges are then all equal to T_r, hence (3) is satisfied. Here, the resultant t will not violate (4) either.

By doing so, we obtain a better initialization within the same complexity, $O(|V||E|)$. The new procedure also justifies the validity of the established correctness proof in Section 4.3.

4.4.2 Multiple r increases

Another factor that affects the efficiency of the algorithm is the convergence rate of r to r^*. In the base algorithm in Figure 3, when $\theta = 0$, there will be exactly one vertex v whose

```
INIT(G, r, T)
  w_r(u,v) ← w(u,v) + r(v) − r(u), ∀(u,v) ∈ E;
  t(v) ← 0, ∀v ∈ V;  Q ← V;
  While (Q ≠ ∅) do
    u ←dequeue(Q);
    For each (u,v) ∈ E do
      If ((w_r(u,v) = 0) ∧ (t(v) < t(u) + d(u,v))) then
        t(v) ← t(u) + d(u,v);
        Q ← Q ∪ {v} if v ∉ Q;
      If (w_r(u,v) > 0) then
        If (t(u) ≤ T) then
          If (t(v) < t(u) + d(u,v) − w_r(u,v)T) then
            t(v) ← t(u) + d(u,v) − w_r(u,v)T;
            Q ← Q ∪ {v} if v ∉ Q;
        Else if (t(v) < d(u,v) − (w_r(u,v) − 1)T) then
          t(v) ← d(u,v) − (w_r(u,v) − 1)T;
          Q ← Q ∪ {v} if v ∉ Q;
```

Figure 6: Pseudocode of initialization procedure with T

$r(v)$ is increased by 1. Although it is necessary, the resultant $T^*_{r'}$ under the new r' is not necessarily better. Generally speaking, r increases on multiple vertices are required in order to reach a better clock period. If we can find an effective way to quickly determine those increases, we can save the iterations in between and expect a dramatic speed-up on practical running time.

Inspired by Lemma 3, we have a similar result.

Lemma 4 *If we initialize t with T^{opt}, and find a vertex $v \in V$ whose $t(v) \geq T^{\text{opt}}$, then there exists a path $p = u \rightsquigarrow v$ such that $t(u) = 0$ and all flip-flops on p have arrival times at least T^{opt}, we can increase $r(v)$ by 1.*

Based on this, if the clock period, obtained in line 23 in Figure 3, is no smaller than T^{opt}, we can initialize t with T^{opt} and see if $t(v) \geq T^{\text{opt}}$ for some $v \in V$. If it is the case, we immediately increase $r(v)$ by 1 and go directly back to line 20 for an r adjustment. Thus, the iterations, which are otherwise required in the base algorithm to identify the v for r increase, are saved here.

4.4.3 Additional termination criterion

One useful information that has not yet been explored is the history of the r updates. According to Lemma 3 and 4, if $t(v) \geq T^{\text{opt}}$ for some $v \in V$, there exists a path $p = u \rightsquigarrow v$ such that $d(p) \geq (w_r(p)+1)T^{\text{opt}}$. We then increase $r(v)$ by 1. Furthermore, there is a stronger relationship between u and v: assuming the optimal clock period has not been reached, if $r(u)$ is increased, then $r(v)$ needs to be increased as well in order to reach the least optimal retiming. This can be easily justified because otherwise the resultant clock period will be larger than the optimum.

Similar to Zhou's idea in [19] to construct a "safe-guard" label, we use $m : V \rightarrow V$ to represent the above relationship. More specifically, for all $v \in V$, $m(v)$ is first set to 0 at the beginning. Later in the algorithm, whenever $r(v)$ is increased, we identify the aforementioned particular u for v and set $m(v) = u$. If the succeeding r adjustment increases $r(x)$ for some $x \in V$, we also set $m(x) = u$. Then we have the following theorem.

Theorem 5 *If the m label forms a cycle, we can terminate the algorithm.*

Proof: Suppose the last m assignment is $m(v) = u$, adding which the m label forms a cycle. Assume the assumption for adding $m(v) = u$ is true, that is, we have not reached the optimal clock period yet. Then, by the definition of m, r will be increased along the cycle and finally result in $r(v) > N_{\text{ff}}$, which by Theorem 3 implies that we have reached the optimal clock period, a contradiction. Therefore, what is in T^{opt} and r^{opt} is already the desired optimal solution. □

It thus provides another criterion to terminate the algorithm. Furthermore, since maintaining m label and checking m-cycle will not increase the asymptotic complexity of the base algorithm, its inclusion will not affect the result in Theorem 4.

To further improve the efficiency, we will first test t at the beginning of each iteration to see if $t(v) = T$ for some $v \in V$. Unnecessary topological sort and succeeding θ computations are thus saved.

5 Experimental results

We implemented the algorithm with the proposed speed-up techniques in a PC with a 2.4 GHz Xeon CPU, 512 KB 2nd level cache memory and 1GB RAM. In order to give a comparison with the results of our previous works [13, 14], we use the same test files, which are modified from ISCAS-89 suite with random delay assignment—1.0 and 2.0 units to gates (treated as macro-blocks) and 0.2 to 5.0 to wires. To reflect the impact of global interconnects in an SOC design, the delay range is intentionally chosen in order for the wire delay to be commensurate or even many times larger than the block delay.

Although we treat gates as blocks, they can only be complete bipartite as all gate has only one output. To further test the cases with non-complete bipartite (denoted as "non-CB" in Table 1 and 2) blocks, we apply hMETIS [10] to partition a circuit into groups. All edges inside a group are then treated as forbidden edges. For simplicity, we did not further apply our timing model to the partitions when generating the results. The number of partitions of a circuit, which is denoted as "No.Part" in Table 1, plays a key role in determining the percentage of non-complete bipartite blocks. To better reflect the influence of non-complete bipartite blocks on the optimal clock period, we intensively choose these numbers in order for the resultant difference to be significant.

In Table 1, we report the computed minimal clock period for each circuit. They match the results in both [13] and [14]. The lower bound ρ^* defined in Lemma 2 is also reported as a comparison. Since we approach the optimal clock period by gradual reduction in the algorithm, we also report the number of reductions for each test file in the column of "No.Step". Note that, although we did not change the topology of the circuit after partitioning, we have forced the type of the edges within a group to E_1. The consequent configuration of edges has little to do with that before partitioning. We report them in one table only to share the basic circuit information, such as $|V|$, $|E|$, N_{ff} and ρ^*, and to be consistent with the report format in [13, 14] for clear comparisons.

In Table 2, we highlight the difference of running time among the algorithms in [13], [14] (both with precision as 0.1) and ours, denoted as t_{bs1}, t_{bs2} and t_{new}, respectively. The results clearly show that our new approach achieves multiple-order improvement over [13] and also generally better than [14]. It confirms that the bound in Theorem 4 is loose.

6 Conclusions

A polynomial-time algorithm for optimal wire retiming is presented in this paper. Contrary to all previous algorithms

Table 1: Minimal Clock Period

Circuit	$\|V\|$	$\|E\|$	N_{ff}	ρ^*	w/o non-CB blocks		w/ non-CB blocks		
					No.Step	T^{opt}	No.Part	No.Step	T^{opt}
s386	519	700	6	51.0	13	51.1	50	1	55.0
s400	511	665	21	32.2	120	32.2	50	1	50.6
s444	557	725	21	35.0	289	35.2	40	1	63.2
s838	1299	1206	32	76.0	2	76.0	130	1	84.0
s953	1183	1515	29	60.6	31	60.6	110	2	69.5
s1488	2054	2780	6	70.1	11	70.6	200	1	73.3
s1494	2054	2792	6	76.8	63	76.9	160	1	79.9
s5378	7205	8603	179	111.0	26	111.2	500	1	115.3
s13207	19816	22999	669	239.5	129	239.5	1000	1	292.8
s35932	46097	58266	1728	148.3	68	148.3	2000	1	163.2
s38584	53473	66964	1452	203.9	126	204.0	2000	1	264.0

Table 2: Running Time Comparison (Seconds)

Circuit	w/o non-CB blocks			w/ non-CB blocks		
	t_{bs1}	t_{bs2}	t_{new}	t_{bs1}	t_{bs2}	t_{new}
s386	1.97	0.01	0.00	3.67	0.01	0.00
s400	1.64	0.01	0.03	3.38	0.01	0.00
s444	2.23	0.03	0.09	4.31	0.01	0.00
s838	8.79	0.03	0.00	33.42	0.02	0.00
s953	9.76	0.04	0.02	17.56	0.07	0.00
s1488	35.17	0.08	0.08	98.88	0.05	0.00
s1494	34.13	0.08	0.06	62.86	0.09	0.00
s5378	684.6	0.24	0.31	1344.74	0.29	0.00
s13207	-	1.07	3.46	-	206.52	0.02
s35932	-	18.63	7.55	-	6.19	0.19
s38584	-	7.44	30.17	-	21992.67	0.19

which used binary search to check the feasibility of a range of clock periods, the new algorithm directly checks the optimality of the current feasible clock period, and can thus either pushes down the period or certifies the optimality.

The underlying idea looks into the incompetent nature of the binary search approach in the continuous domain. At each step, the binary search gives the answer to the question *if the current clock period is feasible*. The optimality of a feasible clock period can only be established indirectly, that is, through the infeasibility of the next smaller clock period. If there is no next smaller clock period as in the continuous domain, the optimality will never be established in theory. However, in our algorithm, the question being answered at each step is *if a feasible clock period smaller than the current one exists*. Since it gives the existence answer, the optimality is established directly once we reach such a step that gives the answer: "No".

Besides the difference of program methodology, our algorithm has many other advantages over the binary search approach. First of all, it is polynomial time bounded. No precision is required. Second, the implementation is simpler. No upper and lower bounds are needed. It is even automatically determined by the algorithm itself how far a necessary step can proceed. Third, the algorithm is very efficient in practice, which is confirmed by the experimental results. Last, but no the least, without using binary search, our algorithm is essentially incremental and has the potential of being combined with other optimization techniques, such as gate sizing, budgeting, etc., thus can be used in incremental design methodologies [4].

References

[1] S. M. Burns. *Performance analysis and optimization of asynchronous circuits*. PhD thesis, California Institute of Technology, Computer Science Department, 1991.

[2] C. Chu, E. F. Y. Young, D. K. Y. Tong, and S. Dechu. Retiming with interconnect and gate delay. In *ICCAD*, pages 221–226, 2003.

[3] P. Cocchini. Concurrent flip-flop and repeater insertion for high performance integrated circuits. In *ICCAD*, pages 268–273, 2002.

[4] J. Cong, O. Coudert, and M. Sarrafzadeh. Incremental CAD. In *ICCAD*, 2000.

[5] J. Cong and S. K. Lim. Physical planning with retiming. In *ICCAD*, pages 2–7, November 2000.

[6] A. Dasdan, S. S. Irani, and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio. In *DAC*, pages 37–42, 1999.

[7] R. B. Deokar and S. S. Sapatnekar. A fresh look at retiming via clock skew optimization. In *DAC*, pages 310–315, 1995.

[8] S. Hassoun and C. J. Alpert. Optimal path routing in single and multiple clock domain systems. In *ICCAD*, pages 247–253, 2002.

[9] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. *JACM*, 44(1):148–199, 1997.

[10] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning:application in vlsi domain. In *DAC*, pages 526–529, 1997.

[11] K. N. Lalgudi and M. C. Papaefthymiou. An efficient tool for retiming with realistic delay modeling. In *DAC*, San Francisco, CA, June 1995.

[12] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Advanced Research in VLSI: Proc. of the Third Caltech Conf.*, pages 86–116. Computer Science Press, 1983.

[13] C. Lin and H. Zhou. Retiming for wire pipelining in system-on-chip. In *ICCAD*, pages 215–220, 2003.

[14] C. Lin and H. Zhou. Wire retiming for system-on-chip by fixpoint computation. In *DATE*, pages 1092–1097, 2004.

[15] P. Pan, A. K. Karandikar, and C. L. Liu. Optimal clock period clustering for sequential circuits with retiming. *IEEE TCAD*, 17(6):489–498, June 1998.

[16] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *ICCAD*, pages 226–233, 1994.

[17] T. Soyata, E. G. Friedman, and J. H. Mulligan. Incorporating interconnect, register, and clock distribution delays into the retiming process. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 16(1):105–120, January 1997.

[18] D. K. Y. Tong and E. F. Y. Young. Performance-driven register insertion in placement. In *ISPD*, pages 53–60, 2004.

[19] H. Zhou. A new efficient retiming algorithm derived by formal manipulation. In *Workshop Notes of Intl. Workshop on Logic Synthesis*, 2004.

[20] H. Zhou, D. F. Wong, I-M. Liu, and A. Aziz. Simultaneous routing and buffer insertion with restrictions on buffer locations. DAC, 1999.