A New Efficient Retiming Algorithm Derived by Formal Manipulation

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

Abstract

A new efficient algorithm is derived for the minimal period retiming by formal manipulation. Contrary to all previous algorithms, which used binary search on a range of candidate periods to check feasibility, the derived algorithm checks the optimality of a current period directly. It is much simpler and more efficient than previous algorithms. Experimental results showed that it is even faster than ASTRA, an efficient heuristic algorithm. Since the derived algorithm is incremental by nature, it also opens the opportunity to be combined with other optimization techniques.

1 Introduction

Since its creation twenty years ago by Leiserson and Saxe [13], retiming has firmly established its reputation as one of the most effective techniques for sequential circuit optimization. The past twenty years have seen retiming's steady improvements on performance and continuous expansions into new areas [14, 18, 16, 20, 21, 5, 15, 17]. Recent progresses on semiconductor technology saw an increase on the number of global wires whose delays are longer than one clock period [10, 1], and retiming is again a promising technique that could be leveraged.

In this paper, we solve the retiming problem by algorithm derivation (also known as *program derivation*) that was advocated and pioneered by Dijkstra [4] among many others. We have two purposes in mind when writing this paper: first, it records a new angle to look at the retiming and a new algorithm for the minimal period retiming problem; second, using the retiming as an example, we hope to bring to the awareness of CAD researchers the advantages of algorithm derivation. For the second purpose, we also give a brief introduction to program derivation, and use the network maximal flow problem to demonstrate the methodology and principles.

Given a sequential circuit, the retiming changes the locations of flip-flops (registers) in the circuit without changing its function. Its validity is guaranteed by the basic operation of moving flip-flops from the inputs to the outputs of a gate, or vice versa. In this paper, we only focus on the minimal period retiming problem, that is, moving the flip-flops to minimized the clock period that is decided by the longest delay between two consecutive flip-flops. Since Leiserson and Saxe [13], the minimal period retiming problem was always solved through a sequence of fixed period retiming problems each of which checks whether a given clock period is feasible. With a list of candidate clock periods or an upper-bound and a lower-bound, a binary search is used to find the smallest feasible period. In cases when the periods may change continuously, the binary search approach only gives a fully polynomial-time approximation scheme (FPTAS) [12], that is, the running time is dependent on the required precisions.

We did not expect any new result when we set up to derive an algorithm for the minimal period retiming problem. But the first surprise is the discovery that neither the fixed period retiming problem nor the binary search comes naturally in the derivation, or we can say that they never come into the picture during our derivation. The derived algorithm iteratively shortens the longest combinational path in the circuit, and when that can no longer be done, certifies that an optimum has been reached. To those who are familiar with Ford and Fulkerson's maximal flow algorithm [7], this sounds too familiar: an incremental flow is attempted over the current flow, and when that is not possible, an optimum is declared. This philosophy is quite different from that of the binary search on fixed period retiming. The main question answered in each step of the binary search approach is *whether a given clock period is feasible*, but the main question in each step of our derived algorithm is *whether any smaller clock period is achievable.* Because of this subtle difference, the optimality of a feasible clock period in the binary search approach can be established only indirectly, that is, through the infeasibility of the next smaller period. However, in the derived algorithm, the optimality of the current clock period can be certified directly through the unattainable of any smaller period.

Compared with the binary search approach, the derived algorithm has many advantages. First, it is very simple. No upper-bound, lower-bound, or list of candidate periods needs to be computed. It does not require any special data structure or subroutine, not even a sorting. Actually, the implementation of the algorithm took us less than two hours– much less than the time we spent on the data preparation. Second, the algorithm is also very efficient, in the sense that all the effects of previous operations are kept. Also, an operation in the algorithm is either pushing down the current clock period or building up evidences to show that it cannot be reduced. It is proved that the derived algorithm has polynomial running time in the worst case. However, the worst case running time should not be our focus, since the running time discrepancy of an algorithm is usually very large and any attempt to reduce the discrepancy will usually increase the running time on some instances while reducing it on others. The derived algorithm will process each instances as efficiently as possible. In the same vein, Shenoy

and Rudall [20] preferred some algorithms with larger worst case running time because of their practical efficiency. Last, but not the least, without using binary search, the derived algorithm is incremental in nature. Because it always keeps a valid retiming during the execution, it has great potentials to be combined with other optimization operations, or used in incremental design methodologies [2].

The rest of the paper is organized as follows. Since we use the predicate calculus and Dijkstra's guarded commands to conduct algorithm derivations, a brief introduction to them is first given in Section 2. As an example, the algorithm derivations for the maximal flow problem are also included in the section. In Section 3, a brand new algorithm in guarded commands is derived for the minimal period retiming problem. In Section 5, the derived algorithm for retiming is translated into a common language where the nondeterminacies in the guarded commands are explored. Experimental results comparing the derived algorithm with other algorithms are given in Section 6. Some conclusions are drawn at the end.

2 Algorithm derivation in guarded commands

2.1 Guarded commands and predicate calculus

Algorithm derivation (or program derivation) is a formal method for developing algorithms. Dijkstra [4] is a classical reference in this area and the guarded commands [3] are usually used in the program derivation.

The language of the guarded commands mainly has four kinds of statements: assignment, composition, selection, and repetition. An assignment statement is of the form

```
v1,v2,...  := E1,E2,...
```

which concurrently assigns the value of each expression on the right hand side to the different corresponding variable on the left hand side. Given two statements `S1, S2`, a composition is the statement `S1; S2` that executes `S1` followed by `S2`.

A guarded command has the following form.

```
<boolean expression>→<statement>
```

The statement at the right of the arrow could be a composite statement. A set of guarded commands can be used to form a selection statement.

```
if <guarded command>{□<guarded command>} fi
```

When more than one guards in a selection statement are true, any statement after a true guard may be selected to execute. This introduces nondeterminacy. When no guard is true, a selection statement is defined as abort. The other way to organize guarded commands is by a repetition statement, which is defined as follows.

```
do <guarded command>{□<guarded command>} od
```

Whenever there is any true guard in the repetition, a statement after any true guard may be executed. This is repeated until all the guards are false. As we can see, nondeterminacy is also allowed here.

The benefit of guarded commands in algorithm derivation is the clean formal definition of their semantics [4, 3]. Based on Floyd [6] and Hoare [11], the semantics of a statement S is defined to truthify a predicate R upon a given predicate P. And this is represented as a Hoare triple:

```
{P} S {R}
```

The predicate calculus [9] is used to express predicates in the algorithm derivation. It has the usual syntax of the first order logic. The only difference is on quantification. The general form of a quantification over $\star$ is exemplified by

$$(\star x, y : R : P),$$

where x and y are distinct index variables, R is a predicate that gives the ranges of x and y, and P is an expression on which $\star$ is applied. The universal and existential quantifications in logic are thus represented as

$$(\forall x :: P(x)) \text{ or } (\wedge x :: P(x)),$$

$$(\exists x :: P(x)) \text{ or } (\vee x :: P(x)),$$

respectively. Besides logic quantifications, other quantifications such as a summation can also be represented similarly. For example, a summation can be put as $\sum_{0\le i\le n} a_i^n$ in the traditional representation. However, it is not clear from the representation which identifier, i or n, is the variable. But when the summation is given as $(+i : 0 \le i \le n : a_i^n)$, it is very clear that i is the index variable and n is the parameter.

A problem can be formally specified by the predicate that the variables must satisfy when the program terminates. This predicate is usually called the post-condition of the program. The algorithm derivation is a goal-driven activity that studies the post-condition and finds a sequence of statements to fulfill it. Besides the program, the intermediate predicates between statements will also be decided in the derivation. Therefore, the proof of the correctness of a algorithm is developed hand-in-hand with the program.

It should be noted that any non-trivial algorithm must involve with at least one repetitive statement–otherwise the processing length of the algorithm will not be longer than the program length and thus cannot do too much. Therefore, a critical task in the algorithm derivation is to partition the post-condition and to decide which part should be kept as an invariant and which part should be fulfilled through the repetition. This will become clear in the actual derivation.

2.2 Derivation for maximal flow

Given a flow network–that is a directed graph $G = (V, E)$ where each $e \in E$ has a capacity $c.e \in \mathcal{R}^+$ and two special nodes $s, t \in V$, the maximal flow problem asks for a flow from s to t that is maximal. It is a well-known problem and has many applications. Ford and Fulkerson [7] were the first to give an algorithm for it. But new efficient algorithms based on the "push-relabel" approach were not discovered until 1988 [8]. To demonstrate the general principle of algorithm derivation, we will derive these two different algorithms with the same methodology.

A flow is defined as a labeling of the edges $f : E \to \mathcal{R}$ that satisfies the following two conditions.

$$\begin{aligned} P0(f): &\quad (\forall e : e \in E : 0 \le f.e \le c.e) \\ P1(f): &\quad (\forall v : v \in V - \{s,t\} : f(-,v) = f(v,-)) \end{aligned}$$

where

$$\begin{aligned} f(-,v) &= (+u : (u,v) \in E : f(u,v)), \\ f(v,-) &= (+w : (v,w) \in E : f(v,w)). \end{aligned}$$

The value of a flow f is defined as the sum of flows out of s, that is

$$|f| = f(s,-)$$

A maximal flow f is a flow that has the maximal value, that is, it must satisfy the following condition.

$$P2: \quad (\forall f' : P0(f') \wedge P1(f') : |f'| \leq |f|)$$

Among the three conditions, $P2$ is the most complex one. Different treatments of it lead to different algorithms.

2.2.1 $\neg P2$ as loop condition

Based on the principle of using the easily satisfied conditions as invariant, we select $P0$ and $P1$ to be invariant, since a simple initialization of

```
f := 0;
```

will truthify them. In order to use $\neg P2$ as the loop condition, we need a witness for it. The negation of $P2$ is

$$(\exists f' : P0(f') \wedge P1(f') : |f'| > |f|).$$

Since both f and f' satisfy $P1$, their difference $f' - f$ also satisfies $P1$. However, the difference may not satisfy $P0$. But the following result can be proved.

Lemma 1

$$(\forall f' : P0(f') : (\forall e : e \in E : -f.e \leq (f' - f).e \leq (c - f).e))$$

Therefore, if $\neg P2(f)$, we must have at least one path $P(f)$ from s to t such that any forward edge e on P has $c.e > f.e$ and any backward edge $e1$ on P has $f.e1 > 0$. Using $(\exists P(f))$ as the loop condition, we can have the following algorithm for the maximal flow problem.

```
f := 0;
{P0∧P1}
do
   (∃P(f)) → {(∃P(f))∧P0∧P1}
             (e:e∈P(f):f.e:=f.e+|P(f)|) {P0∧P1}
od
{P0∧P1∧P2}
```

where $|P(f)| = (\min e : e \in P(f) : c.e - f.e) \min(\min e : -e \in P(f) : f.e)$. Recall that this is Ford-Fulkerson's algorithm.

2.2.2 $P2$ as invariant

The same heuristic may select $P2$ as invariant, since another simple assignment

```
f := 0;
(v:(s,v)∈E:f(s,v):= c(s,v))
```

will truthify $P0 \wedge P2$. Therefore $\neg P1$ will be used as the loop condition. For any node $v \in V - \{s, t\}$, we define

$$X(v) = f(-, v) - f(v, -).$$

With the given initialization, the following predicate is true and we also plan to maintain it as an invariant.

$$I0: \quad (\forall v : v \in V - \{s, t\} : X(v) \geq 0)$$

Therefore, $X(v) > 0$ is a witness of $\neg P1$. Thus, our algorithm will have the following structure.

```
f,X := 0,0;
(v:  (s,v)∈E: f(s,v),X.v := c(s,v),X.v+c(s,v));
do
   X(v)>0 → S;
od
```

It is obvious that S should falsify $X(v) > 0$ but maintain $P0$ and $P2$. That is, it should balance the flows to and from v under the condition of $P0$ and $P2$. There two operations to balance the flows on v: pushing a flow through forward edge (v, w) such that $f(v, w) < c(v, w)$; pushing a flow through backward edge (u, v) such that $f(u, v) > 0$. It seems that the first operations should be selected first since the second is a cancellation of previous operations. However, in the worst case, both operations are necessary to balance the flows on v. But then it may run into the trap of pushing flows back and forth over an edge. Furthermore, we should push a flow back on an edge (s, v) only when we know that $P2$ could be maintained. Both problems indicate that our original plan, that is, falsifying $X(v) > 0$ in one step, is too ambitious.

The possible trap of pushing flows back and forth urges us to introduce an order on the nodes and to only push flows according to the order. Also, $X(v) > 0$ should be falsified gradually: if there are accommodating neighbors of lower order, flows will be pushed out; otherwise, the order of v will be increased. The condition $P2$ will finally be truthified if the order of s is fixed, since in the worst case all flows can be returned to s. The revised algorithm is given as follows, where $h : V \to \mathcal{N}$ is the order of the nodes.

```
f,X,h := 0,0,0;
(v:  (s,v)∈E: f(s,v),X.v := c(s,v),X.v+c(s,v));
h.s := x;
do
   X.v>0∧f(v,w)<c(v,w)∧h.w<h.v →
      f(v,w):=min(c(v,w),f(v,w)+X(v))
[] X.v>0∧f(w,v)>0∧h.w<h.v →
      f(w,v):=max(0,f(w,v)-X(v))
[] X.v>0∧(∀w:f(v,w)<c(v,w)∨f(w,v)>0:h.w≥h.v) →
      h.v:=h.v+1
od
```

In order to decide what value x should be, we need to establish the relation between each edge accommodating a flow and the orders of its two end nodes. It can be shown that the following predicate is an invariant.

$$I1: \quad (\forall u, v : f.(u, v) < c.(u, v) : h.u \leq h.v + 1) \\ \vee (\forall u, v : f.(u, v) > 0 : h.v \leq h.u + 1)$$

Since $h.t = 0$ will not be changed, with $I1$ as an invariant, we can see that a node $u \neq s$ does not have an augmenting path to t if $h.u \geq |V| - 1$. Therefore, we can select $x = |V| - 2$. With the shorthand

$$I = P0 \wedge P2 \wedge I0 \wedge I1$$

the complete algorithm is given as follows.

```
f,X,h := 0,0,0;
(v:  (s,v)∈E: f(s,v),X.v := c(s,v),X.v+c(s,v));
h.s := |V|-2;
{I}
do
   X.v>0∧f(v,w)<c(v,w)∧h.w<h.v →
      {I∧guard} f(v,w):=min(c(v,w),f(v,w)+X.v) {I}
[] X.v>0∧f(w,v)>0∧h.w<h.v →
      {I∧ guard} f(w,v):=max(0,f(w,v)-X.v) {I}
[] X.v>0∧(∀w:f(v,w)<c(v,w)∨f(w,v)>0:h.w≥h.v) →
      {I∧guard} h.v := h.v+1; {I}
od
{I∧(∀v:v∉{s,t}:X.v≤0)}
```

Using $I1$, we can prove that $(\forall u : u \in V : h.u \leq 2|V| - 4)$. This guarantees the termination of the algorithm. Notice that this is Goldberg's "push-relabel" algorithm.

3 A new algorithm for retiming

Circuit retiming is perhaps the most effective structural optimization technique for sequential circuits. It moves the registers within a circuit without changing its function. The minimal period retiming problem needs to minimize the longest delay between any two consecutive registers, which decides the clock period.

The problem can be formally described as follows. Given a directed graph $G = (V, E)$ representing a circuit–each node $v \in V$ represents a gate and each edge $e \in E$ represents a signal passing from one gate to another–with gate delays $d : V \to \mathcal{R}^+$ and register numbers $w : E \to \mathcal{N}$, it asks for a relocation of registers $w' : E \to \mathcal{N}$ such that the maximal delay between two consecutive registers is minimized.

To guarantee that the new registers are actually a relocation of the old ones, a label $r : V \to \mathcal{Z}$ is used to represent how many registers are moved from the outgoing edges to the incoming edges of each node. Using this notation, the new number of registers on an edge (u, v) can be computed as $w'(u, v) = w(u, v) + r.v - r.u$. Furthermore, to avoid explicitly enumerating the paths, we introduce another label $t : V \to \mathcal{R}^+$ to represent the output arrival time of a gate, that is, the maximal delay of the gate from any preceding register. Based on the notations, the validity of a retiming (r, t) is defined by the following conditions.

$$\begin{aligned} P0(r): &\quad (\forall (u,v) \in E :: w(u,v) + r.v - r.u \geq 0) \\ P1(t): &\quad (\forall v \in V :: t.v \geq d.v) \\ P2(r,t): &\quad (\forall (u,v) \in E : r.u - r.v = w(u,v) : t.v - t.u \geq d.v) \end{aligned}$$

We use a predicate P to denote the conjunction of the above conditions:

$$P(r, t) \triangleq P0(r) \wedge P1(t) \wedge P2(r, t)$$

The optimality of a retiming (r, t) is given by the following condition.

$$P3: \quad (\forall r', t' : P(r', t') : \max.t \leq \max.t')$$

where

$$\max.t \triangleq (\max v : v \in V : t.v).$$

Since we only talk about a valid retiming (r', t') in the sequel, to simplify the presentation, we often omit the range condition $P(r', t')$; the meaning will be clear from the context.

The condition $P0$ states that a valid retiming should have non-negative number of registers on any edge. The conditions $P1$ and $P2$ defines a lower bound on the arrival time t, that is, the arrival time of a gate is at least the summation of the gate delay and the arrival time of its fanins. The condition $P3$ states that among all valid retimings–those satisfy $P0$, $P1$, and $P2$–, this current (r, t) has a minimal $\max.t$.

Similar to the conditions of the maximal flow problem, $P3$ gives the optimality condition and is the most complex one. Based on the same heuristic, we consider a simple initialization as follows.

```
r,t:=0,d;
do
   (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
      t.v:=t.u+d.v
od
```

This initialization will truthify $P0$, $P1$, and $P2$. To use $\neg P3$ as a loop condition, we need to establish a witness for it. If $\neg P3$, that is, we have another valid retiming (r', t') such that $\max.t > \max.t'$, then $(\forall v : t.v = \max.t : t'.v < t.v)$. One property we know for these nodes is

$$(\forall v : t'.v < t.v : (\exists u : t.u = d.u : r.u - r.v > r'.u - r'.v)),$$

which means that if the arrival time of v is smaller in another retiming (r', t'), then there must be a node u such that r' has more registers between u and v. In fact, one such a u is the starting node of the longest combinational path to v that gives the delay of $t.v$. It should be noted that it is not the absolute values of r but their differences that are relevant in the retiming. If (r, t) is a solution to a retiming problem, then $(r+c, t)$, where $c \in \mathcal{Z}$ is an arbitrary constant, is also a solution. Therefore, we can move r "closer" to r' by allocating more registers between u and v, that is, by either decreasing $r.u$ or increasing $r.v$. We know that v can be easily identified by $t.v = max.t$. In order to find u, we will keep yet another label $p : V \to V$ such that $p.v$ is the starting node of the longest combinational path to v for any $v \in V$. No matter whether $r.v$ or $r.p.v$ is selected to change, the amount of change should be only 1 since we do not want to over-adjust r. It means that, after the adjustment, we still have $r.v - r.p.v \leq r'.v - r'.p.v$, or equivalently $r.v - r'.v \leq r.p.v - r'.p.v$. Assume we increase $r.v$. The arrival time $t.v$ can be immediately reduced to $d.v$. This operation is given by the following guarded command.

```
(∃r',t'::max.t'<max.t)∧t.v=max.t →
   r.v,t.v,p.v:= r.v+1,d.v,v
```

Since registers are moved in the above operation, the condition $P2$ may be violated. To restore it, we may execute the same repetition statement as in the initialization after each operation, as in the following form.

```
(∃r',t'::max.t'<max.t)∧t.v=max.t →
   r.v,t.v,p.v:= r.v+1,d.v,v;
   do
      (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
         t.v,p.v:=t.u+d.v,p.u
   od
```

However, this kind of programming will aggressively update t after each adjustment of r, and its only purpose is to keep $P2$ invariant when r is changed. Alternatively, we can weaken the invariant to be maintained, and allow $P2$ to be violated temporally and restored later. This can be done by putting the two guarded command within the same repetition statement; it increases the flexibility in their execution orders.

```
   (∃r',t'::max.t'<max.t)∧t.v=max.t →
      r.v,t.v,p.v:= r.v+1,d.v,v
[] (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
      t.v,p.v:=t.u+d.v,p.u
```

The execution of the second guarded command will increase t. If we use maxT to represent the $\max.t$ before we adjust r, very likely, such t increases may cause $t.y \geq$ maxT for some $y \in V$. Similarly, based on the assumption $(\exists r', t' :: \max.t' <$ maxT$)$, we must have $r.y - r.p.y < r'.y - r'.p.y$. Therefore $r.y$ should also be increased. This can be included in the above commands through a simple modification.

```
   (∃r',t'::max.t'<maxT)∧t.v≥maxT →
      r.v,t.v,p.v:= r.v+1,d.v,v
[] (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
      t.v,p.v:=t.u+d.v,p.u
```

The difference between the two cases of increasing r is that in the first case we have $t.v = \max.t$ but in the second case it may not be true. With $t.v = \max.t$, there is no edge (v, x) such that $r.v - r.x = w(v, x)$, and thus the execution of $r.v := r.v + 1$ cannot destroy $P0$. Without it, that is not guaranteed. Similar to our handling of $P2$, we can either maintain $P0$ through a repetitive updating of r after each operation or allow it to be violated temporally and restored later. We select the second option since it renders more flexibility. It gives us one more guarded command in addition to the above two.

```
   (∃r',t'::max.t'<maxT)∧t.v≥maxT →
      r.v,t.v,p.v:= r.v+1,d.v,v
[] (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
      t.v,p.v:=t.u+d.v,p.u
[] (u,v)∈E∧r.u-r.v>w(u,v) →
      r.v,t.v,p.v:=r.u-w(u,v),t.u+d.v,p.u
```

The condition $\neg P3$, that is $(\exists r', t' :: max.t' < \text{maxT})$, guarantees that the above iterative operations to push down $t.v \geq \text{maxT}$ will terminate within finite steps. This comes from the fact that *each time after $r.v$ for any $v \in V$ is increased, it is guaranteed that there exists a $u \in V$ such that $r.u - r.v \geq r'.u - r'.v$, or equivalently $r.u - r'.u \geq r.v - r'.v$.* Therefore $(\max v : v \in V : r.v - r'.v)$ cannot be increased during the operations. When the iterations terminate, we will have a valid retiming (r, t) such that $\max.t < \text{maxT}$. Therefore, we can reset the maxT and start the process again. Once again, we introduce a guarded command parallel to the above three instead of introducing hierarchy. The algorithm currently has the following scheme.

```
r,t,p,maxT := 0,d,1,0;
do
   (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
      t.v,p.v := t.u+d.v,p.u
[] maxT<t.v → maxT:=t.v
od
{P(r,t)∧max.t=maxT}
do
   (∃r',t'::max.t'<maxT)∧t.v≥maxT →
      r.v,t.v,p.v:= r.v+1,d.v,v
[] (u,v)∈E∧r.u-r.v≥w(u,v)∧t.v<t.u+d.v →
      t.v,p.v := t.u+d.v,p.u
[] (u,v)∈E∧r.u-r.v>w(u,v) →
      r.v,t.v,p.v:=r.u-w(u,v),t.u+d.v,p.u
[] P(r,t)∧max.t<maxT → maxT:=max.t
od
{P(r,t)∧max.t=maxT∧(∀r',t'::max.t'≥maxT)}
```

The invariant of the second repetitive statement is now very weak–perhaps only includes $P1$; the post-condition comes from the negation of the guards.

The remaining task to complete the algorithm is the calculation of the predicate $(\exists r', t' :: \max.t' < \text{maxT})$. We already know that if it is true then $(\max v : v \in V : r.v - r'.v)$ cannot be increased. This implies that there is at least a node v such that $r.v$ does not change. We use a label $m : V \to V$ for each node v to point to the "safe-guard" node $p.v$ when $r.v$ is increased. Since $r.v - r.p.v + w(p.v, v) = 0$ before the increase (there is no register between $p.v$ and v), we know that

$$(\forall v : m.v \in V : r.v - r.m.v \leq 1)$$

is an invariant, which means that $r.v$ is at most one larger than $r.m.v$. The condition $(\exists r', t' :: \max.t' < \text{maxT})$ guarantees the predicate

$$(\forall v : m.v \in V : r.m.v - r'.m.v \geq r.v - r'.v),$$

which ensures that the label m will not form any cycle. This means that m will form a forest where the roots have $r = 0$ and a child can have a r at most one larger than that of its parent. Therefore, if $(\exists r', t' :: \max.t' < \text{maxT})$, then, for any $0 < i \leq |V|$, there must be at least i nodes whose r are smaller than i. A violation of any of these conditions presents an evidence for $(\forall r', t' :: max.t' \geq \text{maxT})$–that is, maxT is optimal. Therefore, we can simply extend the above scheme with the m pointers and monitor these optimality evidences–that is, $(\exists v :: r.v > |V| - 1) \vee (\forall v :: r.v > 0)$ or m forms a cycle.

The monotonic decrease of maxT implies a monotonic strengthening of the predicate $(\exists r', t' :: \max.t' < \text{maxT})$. In other words, we have

$$\text{maxT}_1 > \text{maxT}_2 \Rightarrow$$
$$((\exists r', t' :: \max.t' < \text{maxT}_2) \Rightarrow (\exists r', t' :: \max.t' < \text{maxT}_1)).$$

It shows that the operations done under a larger maxT_1 is conservative and still valid under a smaller maxT_2, and the conditions given by $(\exists r', t' :: \max.t' < \text{maxT}_1)$ are still true if $(\exists r', t' :: \max.t' < \text{maxT}_2)$. Therefore, we do not need to reset any of r or m after each decrease of maxT. This gives the beauty of the algorithm: it constructively pushes down $\max.t$, and at the same time prepares evidences to show that $\max.t$ is optimal.

Based on the discussion, the complete algorithm is given as follows.

```
r,t,p,m,maxT,cycle:=0,d,1,0,0,0;
do
   (u,v)∈E∧r.u-r.v=w(u,v)∧t.v-t.u<d.v →
      t.v,p.v := t.u+d.v,p.u
[] maxT<t.v → maxT:=t.v
od
{P(r,t)∧max.t=maxT}
do
   ¬cycle∧t.v≥maxT →
      if
         m.v≠0 → cycle:=(m forms a cycle)
      [] m.v=0 → skip
      fi;
      r.v,t.v,m.v,p.v := r.v+1,d.v,p.v,v
[] (u,v)∈E∧r.u-r.v=w(u,v)∧t.v<t.u+d.v →
      t.v,p.v:=t.u+d.v,p.u
[] (u,v)∈E∧r.u-r.v>w(u,v) →
      r.v,t.v,m.v,p.v:=r.u-w(u,v),t.u+d.v,u,p.u
[] P(r,t)∧max.t<maxT → maxT:=max.t
od
{(∃r,t::max.t=maxT)∧(∀r',t'::max.t'≥maxT)}
```

The correctness of the algorithm is readily provable by using the predicate annotations in the program. It should be noted that, since we start to change r before we know $(\exists r', t' :: \max.t' < \text{maxT})$, the post-condition only states that maxT is the optimal period, but not that (r, t) is an optimal retiming. However, an optimal retiming can be easily computed if we store the feasible r before trying to push the current maxT down. In the post-condition, the predicate $(\exists r, t :: \max.t = \text{maxT})$ is an invariant of the loop and the predicate $(\forall r', t' :: \max.t' \geq \text{maxT})$ is implied by *cycle* which comes from the negation of all guards in the loop. The termination is guaranteed by the monotonic increase of r and the upper bound of $|V| - 1$ on them. In order to clear the doubt on the possibility of an inhibitively long running time when each reduction on $max.t$ is too small, a bound on the worst case running time is given in the following theorem.

Theorem 1 *The worst case running time of the derived retiming algorithm is upper bounded by* $O(V^2E)$.

Cautions should be used on this bound. First, a program will usually have great running time variations on different problem instances. The worst case time may only happen in a few rare instances, and thus may not be a good indication of the efficiency on most other instances. Second, even the worst case happens on most problem instances, a bound may be loose due to the difficulty to have an accurate analysis. Since only necessary operations are conducted in each step of the derived algorithm, it should be efficient in most instances. This is confirmed by our experiments.

4 Intuition and example

To help the readers to get a better understanding of the derived retiming algorithm, we give an example to show the intuitive interpretation of the algorithm and why it is efficient in practice. The example is given in Figure 1. The first graph shows the original circuit; it has five gates with delays shown in each gate, and registers are shown on the edges. The steps of the algorithm are shown by the sequence of graphs following the first one.

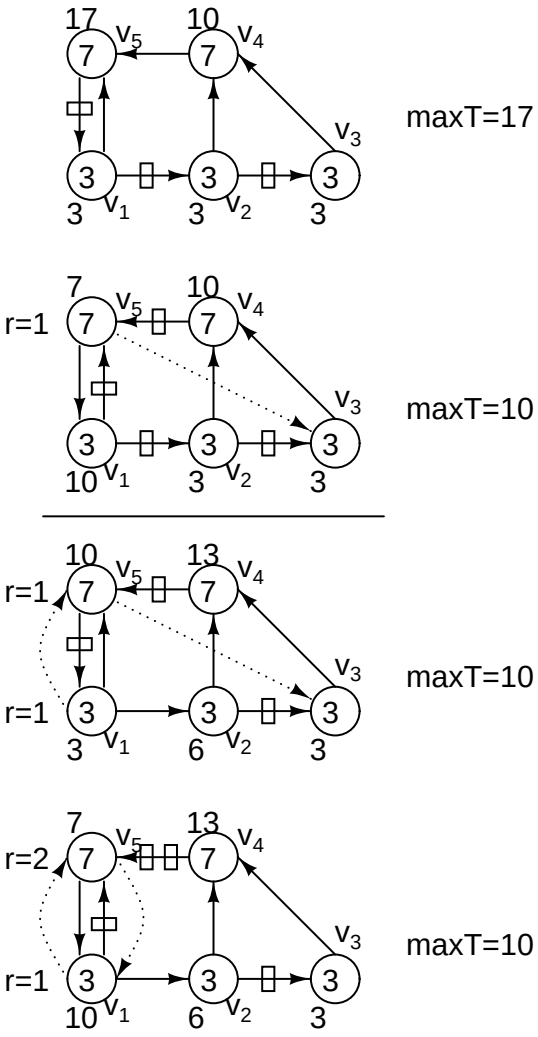

Figure 1: The algorithm applied on an example.

On the original circuit, the algorithm first calculates the arrival time of each gate, which is shown beside each gate. Since the gate v_5 has the largest arrival time, 17, the maxT is set to 17. Then any node whose arrival time is at least maxT will have its r incremented. In our case we have $r(v_5) = 1$, which means a register is moved from the output of v_5 to its inputs. This increment will be accompanied by setting the m pointer $m(v_5) = v_3$ (because v_3 is the starting point of the longest path), which is represented by a dotted edge in the second graph. The arrival time $t(v_5)$ is also updated to 7. And the update is propagated to make $t(v_1) = 10$. Now since max $.t$ is pushed down to 10, we reset maxT to 10 and start another iteration.

Since now there are two nodes v_1 and v_4 whose arrival time is at least maxT, any of them could be picked up for operation. Assume that v_1 is selected. The update is shown in the third graph, where we have $r(v_1) = 1$, $m(v_1) = v_5$, and $t(v_1) = 3$. The arrival time propagation will set $t(v_2) = 6$, $t(v_4) = 13$, and $t(v_5) = 10$. Now we have $t(v_5) = 10$ and $t(v_4) = 13$ which are at least maxT (that is 10). Any of them can be selected for operation. If v_5 is selected, then $r(v_5) = 2$ and $m(v_5) = v_1$, as shown in the last graph. Therefore, a cycle is formed by m between v_1 and v_5. It is an evidence that there does not exist a retiming with a period smaller than maxT = 10. Therefore maxT = 10 is the optimal period and it is realized in the second graph.

A few interesting things can be noticed from the example. The first is the dances of the m pointers accompanying the r increases. Each time when a node has its r increased, its m pointer dances (being created or changed). Before the optimal period is reached, the m pointers can only form a forest. It is also true that an m pointer can only point to a node whose r is at most one smaller. In the last step of the example, when $r(v_5)$ becomes 2, it can only point to v_1 which is the only node with nonzero r. The other thing is that the operations for finding an evidence of optimality are the same as those for improving the solution. The first step in the example reduces the period, while the next two steps discover an optimality evidence. But they have the same operations. Also, the operations that reduce the period usually take shorter time than those that build up evidences, since the former cannot go through a cycle. On this aspect, this derived retiming algorithm is also similar to Ford-Fulkerson's maximal flow algorithm: finding an augmenting path is much faster than discovering its nonexistence. This also explains why the derived algorithm is more efficient in practice: improvement iterations are very fast and only the last iteration that establishes the optimality takes longer time.

5 Implementation

The advantages of the guarded commands include simplicity, symmetry, and intrinsic nondeterminacy. They are of great help in algorithm derivations and correctness proofs. However, to implement the derived algorithm, it needs to be translated into a common programming language such as the C language. We should emphasize that the translation is straight-forward and there is no hiding trick. The translation of the retiming algorithm is presented here for two purposes. First, it may help those who are not familiar with the guarded commands to understand the new retiming algorithm. Second, it demonstrates how the nondeterminacy in guarded commands can be further explored for the benefit of program performance.

The four kinds of statements in the guarded commands

all have directly corresponding constructs in any common programming language such as C. The assignment and composition statements are the same as in C, with the exception that a concurrent assignment may be implemented by multiple sequential assignments in C. The selection statement corresponds to the **if** statement in C, and the repetition statement to the **while** statement. However, nondeterminacy is intended to be allowed in the guarded commands. When more than one guards in a selection or repetition statement are true, any one of the statements guarded by them may be executed. With nondeterminacy, the algorithm design only makes necessary decisions that guarantee the correctness. An algorithm with nondeterminacy in the guarded commands actually represents a set of deterministic algorithms, and performance considerations could be used to make further decisions when it is translated into a deterministic one.

In the new retiming algorithm derived in the previous section, there are two kinds of nondeterminacy: multiple guards could be true in each of the two repetition statements; multiple instances (vertices or edges) may satisfy one given guard. The important fact is that *the algorithm is always correct no matter what execution order is used.* To avoid searching all vertices or edges for instances satisfying the guards, a queue Q is used in the implementation for book-keeping. Assuming there are n gates in the circuit, arrays d and w are used for gate delays and edge weights (the numbers of registers). Similarly the variables r, t, p, m in the derived algorithm are implemented as arrays. Based on the discussion, the C program pseudo-code of the retiming algorithm is given as follows.

```
for (i=0; i<n; i++) {
  r[i] = 0; t[i] = d[i]; p[i] = i; m[i] = -1;
}
for ((i,j)∈E)
  if (w[i][j] == 0 && t[j]<t[i]+d[j]) {
    t[j] = t[i]+d[j]; p[j] = p[i];
    Q = Q+{j};
  }
while (Q!=∅) {
  i = dequeue(Q);
  for ((i,j)∈E)
    if (w[i][j] == 0 && t[j]<t[i]+d[j]) {
      t[j] = t[i]+d[j]; p[j] = p[i]; Q = Q+{j};
    }
  if (maxT<t[i]) maxT = t[i];
}
while (count < n) {
  for (i=0; i<n-1; i++)
    if (t[i] == maxT) Q = Q+{i};
  while (count < n && Q!=∅) {
    i = dequeue(Q);
    if (t[i] >= maxT) {
      if (m[i] != -1) {
        count = 0; j = m[i];
        while (count <n && j != i && m[j] != -1) {
          j = m[j]; count ++;
        }
        if (m[j] != -1) {
          count = n; break;
        }
      }
      r[i] ++; t[i] = d[i]; m[i] = p[i]; p[i] = i;
    }
    for ((i,j)∈E)
      if (r[i]-r[j]==w[i][1] && t[j]<t[i]+d[j]) {
        t[j] = t[i]+d[j]; p[j] = p[i]; Q = Q+{j};
      } else if (r[i]-r[j] > w[i][j]) {
        r[j] = r[i]-w[i][j]; t[j] = t[i] + d[j];
        p[j] = p[i]; m[j] = i; Q = Q+{j};
      }
  }
  if (count<n) {
    maxT = 0;
    for (i=0; i<n; i++)
      if (t[i]>maxT) maxT = t[i];
  }
}
```

The first **for** statement corresponds to the variable initialization in the derived algorithm. The second **for** statement and the first **while** statement correspond to the first repetition statement, which is used to compute t and $\max.t$. The second **while** statement implements the second repetition statement in the derived algorithm. Since the last guard in the statement is mutually exclusive from other guards, we implement the first three guarded commands by the inner **while** statement and the last guarded command by the **if** statement after it. Since the processing of the first guarded command may change the second or the third guard value, it is executed first in the `if (t[i]>=maxT)` statement. Because of these, the statement `r[j] = r[i]-w[i][j]` may have an increase larger than 1, which speeds up the convergence. We implement a cycle checking along m by simply counting and searching for the starting vertex i. Other alternatives are also possible.

Since the order of getting vertices from Q is not defined, there is still some flexibility in the C program. If Q is implemented as a stack–that is, first-in-last-out (FILO), the updates will be conducted in a depth-first fashion; if Q is implemented as a first-in-first-out (FIFO) queue, the updates will be conducted in a breadth-first fashion. Our experiments showed that the depth-first update is a little better on performance.

6 Experimental results

We implemented the derived retiming algorithm very easily according to the code in the previous section. We also got the minimal period retiming code ASTRA [19] from Prof. Sapatnekar. The parser and data preparation (e.g. changing registers into edge weights and adding a host node connecting POs and PIs) in ASTRA are also used with the derived retiming program. It should be noted that ASTRA used the equivalence between retiming and clock skew optimization to first do a continuous retiming and then locally move registers to minimize skews [19]. Therefore, it is a heuristic algorithm and may not give the optimal period if no clock skew is allowed. All the test cases in the ISCAS89 benchmarks are tested both on the derived algorithm and the ASTRA running on a Sun Ultra 10 machine. Since there is no gate delay information on those benchmarks, the ASTRA is set to generate gate delays between 1 and 100. Reported in Table 1 are results for large test cases. For each test cases, it reports its name, number of gates, the original period and the optimal period (from the derived algorithm). The running time of the derived algorithm (column "time") and that of the ASTRA (column "astra") are reported for comparison. For almost all cases, the derived algorithm outperforms the ASTRA. For larger circuits and larger difference between the original and optimal periods, the difference is even bigger. The result is striking since we are comparing an exact algorithm with a heuristic algorithm. Since the ASTRA only

reported the achieved clock period with clock skews but not the amount of required skews, we cannot measure how far away its results are from the optimal.

Table 1: Experimental Results

name	#gates	clock period		time(s)	astra(s)
		before	after		
s1423	490	166	127	0.02	0.04
s1494	558	89	88	0.02	0.01
s9234	2027	89	81	0.12	0.19
s9234.1	2027	89	81	0.16	0.20
s13207	2573	143	82	0.12	0.49
s15850	3448	186	77	0.36	0.57
s35932	12204	109	100	0.28	0.86
s38417	8709	110	56	0.58	1.46
s38584	11448	191	163	0.41	1.12
s38584.1	11448	191	183	0.48	1.26

7 Conclusions

A new efficient algorithm for the minimal period retiming is presented in this paper. Contrary to all previous algorithms 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 period, and can thus either push down the period or certify the optimality. The advantages of the algorithm include its simplicity, efficiency, and being incremental. Experimental results shows that the algorithm is faster even than the best heuristic for the same problem.

Besides the algorithm, the paper also presents the algorithm design methodology by which the algorithm is discovered: program derivation. Our experiences so far with this method are positive and, through this new retiming algorithm, we hope to increase the awareness of CAD researchers on this method.

References

[1] P. Cocchini. Concurrent flip-flop and repeater insertion for high performance integrated circuits. In *Proc. Intl. Conf. on Computer-Aided Design*, 2002.

[2] Jason Cong, Olivier Coudert, and Majid Sarrafzadeh. Incremental CAD. In *Proc. Intl. Conf. on Computer-Aided Design*, 2000.

[3] E. W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *Communications of the ACM*, 8:453–457, 1975.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[5] G. Even, I. Y. Spillinger, and L. Stok. Retiming Revisited and Reversed. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 15(3):348–357, March 1996.

[6] R. W. Floyd. Assigning meanings to program. In *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, volume 19, pages 19–31, 1967.

[7] J. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[8] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[9] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag New York, Inc., 1993.

[10] S. Hassoun and C. J. Alpert. Optimal path routing in single and multiple clock domain systems. In *Proc. Intl. Conf. on Computer-Aided Design*, 2002.

[11] C. A. R. Hoare. An axiomatic basis for computing programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[12] Alexander T. Ishii, Charles E. Leiserson, and Marios C. Papaefthymiou. Optimizing two-phase, level-clocked circuitry. *Journal of the ACM*, 44(1):148–199, January 1997.

[13] C. E. Leiserson and J. B. Saxe. Optimizing Synchronous Systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, Spring 1983.

[14] B. Lockyear and C. Ebeling. Optimal retiming of level-clocked circuits using symmetric clock schedules. In *IEEE Transactions on Computer Aided Design*, volume 13, pages 1097–1109, September 1994.

[15] N. Maheshwari and S. S. Sapatnekar. Optimizing large multi-phase level-clocked circuits. *IEEE Transactions on Computer Aided Design*, 18(9):1249–1264, September 1999.

[16] S. Malik, K. J. Singh, R. K. Brayton, and A. Sangiovanni-Vincentelli. Performance optimization of piplelined circuits using peripheral retiming and resynthesis. *IEEE Transactions on Computer Aided Design*, 12(5):568–578, May 1993.

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

[18] M. C. Papaefthymiou and K. H. Randall. Tim: A timing package for two-phase, level-clocked circuitry. In *Proc. of the Design Automation Conf.*, pages 497–502, Dallas, June 1993.

[19] S. S. Sapatnekar and R. B Deokar. Utilizing the retiming-skew equivalence in a practical algorithm for retiming large circuits. *IEEE Transactions on Computer Aided Design*, 15(10):1237–1248, October 1996.

[20] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 226–233, 1994.

[21] V. Singhal, C. Pixley, R. L. Rudell, and R. K. Brayton. The Validity of Retiming Sequential Circuits. In *Proc. of the Design Automation Conf.*, pages 316–321, San Francisco, CA, June 1995.