Deriving a New Efficient Algorithm for Min-Period Retiming*

Hai Zhou
ECE, Northwestern University, Evanston, IL 60208

Abstract

A new efficient algorithm is derived for the minimal period retiming problem by formal methods. Contrary to all previous algorithms, which used binary search to check feasibilities on a range of candidate periods, 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 [11], 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. Recent progresses on semiconductor technology saw an increase on the number of global wires whose delays are longer than one clock period [8, 1], and retiming is again a promising technique that could be leveraged [14].

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.

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 [11], the minimal period retiming problem has always been solved through a sequence of fixed period retiming problems each of which checks whether a given clock period is feasible. With a list or an interval of candidate clock periods, a binary search is used to find the smallest feasible period. If the candidate periods form a continuous range, the binary search approach only gives a fully polynomial-time approximation scheme (FPTAS) [10], that is, the running time is dependent on the required precision.

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, declares that an optimum has been reached. This philosophy is quite different from that of binary search with 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.

It should be noted that even in a newer algorithm, FEAS, presented by Leiserson and Saxe [12] and adopted later by De Micheli [2] and Evan et al. [5], binary searches with feasible period checking were still used. Therefore, even though the FEAS algorithm looks similar to our derived algorithm, our algorithm is different and better. Furthermore, our algorithm can be easily modified for the initial state computation similar to [5].

2 Algorithm derivation in guarded commands

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 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 [9], 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:

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

{P} S {R}

The predicate calculus [7] 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

$$(\forall x :: P(x)) \text{ and } (\exists x :: P(x)).$$

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 have at least one repetitive statement–otherwise the processing length of the algorithm will not be longer than the program length and thus cannot do 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.

3 Deriving an algorithm for min-period 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 output 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 $P2$–, the current (r, t) has a minimal $\max.t$.

Among all the conditions, $P3$ gives the optimality condition and is the most complex one. Based on the idea of maintaining simple invariants, we consider an initialization as follows to truthify $P0$, $P1$, and $P2$.

```
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
```

Our plan is to have a loop (i.e. repetitive statement) after the initialization to iteratively make $P3$ true while maintaining the other conditions. This means that $\neg P3$ will be used as the loop condition; we will study what can be done under $\neg P3$.

$\neg P3$ means that we have another valid retiming (r', t') such that $\max.t > \max.t'$. Therefore, we have $(\forall v : t.v = \max.t : t'.v < t.v)$. If the longest path l that gives $t.v$ starts from u, then l has no register, or equivalently, $w.l + r.v - r.u = 0$. But since $t'.v < t.v$, there must be at least one register on l in retiming (r', t'), or $w.l + r'.v - r'.u > 0$. This is stated as the following property.

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

It should also 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 decide to 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 the operations, very likely, such t increases may cause $t.y \geq \text{maxT}$ for some $y \in V$. Similarly, based on the assumption $(\exists r', t' :: \max.t' < \text{maxT})$, 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 $r.v - r.u \leq r'.v - r'.u$, or equivalently $r.v - r'.v \leq r.u - r'.u$.* 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 a 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 $r.v - r.p.v + w.l = 0$ before the increase and $w.l \geq 0$ (l is a path from $p.v$ to 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

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

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 prohibitively 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 when the worst case happens, 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 Experimental results

The derived retiming algorithm is implemented easily. The nondeterminacy in the guarded commends is explored by using a queue of modified nodes. For comparison, we also got the minimal period retiming code ASTRA [13] 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 leveraged for 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 [13]. 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 circuit 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

References

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

[2] G. De Micheli. Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(1):63–73, January 1991.

[3] E. W. Dijkstra. Guarded commands, nondeterminacy, and the formal derivation of programs. *CACM*, 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] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag New York, Inc., 1993.

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

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

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

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

[12] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

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

[14] H. Zhou and C. Lin. Retiming for wire pipelining in system-on-chip. *IEEE TCAD*, 23(9):1338–1345, September 2004.