Optimal design of synchronous circuits using software pipelining techniques

François-R. Boyer[1], El Mostapha Aboulhamid[1], Y. Savaria[2] and I.E. Bennour[3]

[1] Université de Montréal, [2] École Polytechnique de Montréal, [3] Nortel

Abstract

In this paper, we present a method to optimize clocked circuits by relocating and changing the time of activation of registers to maximize throughput. Our method is based on software pipelining instead of retiming. The two methods have the same overall complexity but unlike previously published retiming methods, the time consuming step of searching an adequate clock period is avoided, since the optimal clock period is always a solution. The resulting circuit is a multi-phase-clocked circuit, where all the clocks have the same period. Edge-triggered flip-flops are used where the combinational delays exactly match that period, whereas level-sensitive latches are used elsewhere improving the area occupied by the circuit.

Experiments on existing and newly developed benchmarks show a substantial performance improvement compared to previously published work.

Keywords: retiming, software pipelining, sequencial circuits, logic synthesis, resynthesis

1 Introduction

In a synchronous circuit, clocked storage elements are used to regulate the data flow and to provide stable inputs while functions (combinational logic) are evaluated. The speed of the circuit is determined not only by the calculation time of these functions, but also by the time wasted waiting for the synchronization point (clock) to arrive. In this paper we focus on methods for minimizing that wasted time. The proposed method also gives the circuit path that limits the speed, which is where logic should be optimized if we need more speed. For circuits synchronized by a single periodic event (single clock), the wasted time between two storage elements (registers) is the period time minus the combinational delay between these storage elements.

Leiserson and Saxe in [LEI 91] reduced the wasted time by moving registers to minimize the maximum combinational delay between two registers and changing the clock period to that value. This register movement does a better repartition of combinational logic, resulting in a tighter fit in the clock cycles. The method they present, called retiming, is proved to give the register placement that permits the smallest clock period under the constraints that registers are edge-triggered and are all controlled by the same clock. However, it was found that in many cases that solution in not optimal, because registers cannot always be moved so that no time is wasted on the critical path. Indeed, a part of the circuit is always “retimed” by an integral number of periods; with some kind of fractional retiming we could have better results.

Lockyear and Ebeling in [LOC 94] presented an extension to retiming using level-sensitive registers (latches) with multi-phase clocks, the phases being fixed by the designer instead of being computed automatically. The use of multi-phase clocks permits to “retime” a part of the circuit by one phase instead of a whole period, which gives a better resolution and thus a tighter fit to waste less time. In fact, that method will give an optimal solution, with no wasted time on the critical path, but only if the phases specified permit it and if we suppose that we can have a perfect clock.

Deokar and Sapatnekar in [DEO 95] define the "equivalence between clock skew and retiming", which they use to minimize the clock period. They first calculate a "skew" that should be applied to the clock input of each flip-flop in order to have the desired period. Then they apply there equivalence to retime the circuit to bring down the "skews" to zero (or as close as possible). To find the optimal period it uses a binary search, verifying if there is a "skew" that permits to run the circuit with that clock period. If the "skews" are not zero, we can force them to zero (increasing a little the clock period) or add circuitry to do that skew on the clock. It is not clear that the circuit with the skews on the clock will be correct because they ignore the short path constraints.

Maheshwari and Sapatnekar in [MAH 97, MAH 98] use retiming to reduce the area (number of register) for a given clock period. They use a longest path algorithm to find ASAP and ALAP locations of the registers, which permit to reduce the computation time by reducing the size of the LP problem to be solved.

Ishii, Leiserson and Papaefthymiou in [ISH 97] present methods to minimize the clock period on a multi-phase level-clocked circuit. They also show how to convert an edge-clocked circuit into a faster level-clocked one. The method is in two steps: retiming and clock tuning. For a k-phase simple circuit, minimizing the clock period using clock tuning is $O(k V^2)$ and retiming to achieve a given clock period for fixed duty-ratios is $O(k V^3)$. Approximation schemes for solving the two steps at once, to achieve the minimum clock period, are given. For simultaneous retiming and clock tuning with no conditions on the duty cycles on a two-phase circuit, an approximation, with a period at most $(1+\varepsilon)$ times the optimal, can be found in $O(|V|^3 (1/\varepsilon) \log(1/\varepsilon) + (|V| |E| + |V|^2 \log |V|) \log(|V|/\varepsilon))$. For k-phase, the running time for that approximation contains a factor of ε^{-k}, which is impractically large for small values of ε.

Legl, Vanbekbergen and Wang in [LEG 97] extend retiming to handle circuits where not all the registers are enabled at the same time. The idea is that we can move registers across a logic block only if they are enabled by the same signal. They do not change the enable time of registers.

Work has also been done to speedup loops on parallel processors. The software pipelining method discussed in [DON 92] gives an optimal schedule of the operations (with no wasted time on the critical cycle) if there are no constraints on the resources (number of operative units). Also, there are methods that use retiming as an heuristic to find schedules when there are resource constraints [BEN 95].

We present a method that use software pipelining, instead of retiming, to find an optimal schedule of the operations in a circuit, and then a way to reconstruct the circuit from that schedule. Scheduling is much like calculating the clock skews in [DEO 95, MAH 97, MAH 98], but then we do not apply retiming according to that schedule. Like said previously, it is unclear if their circuit works when we do not force the skews to be zero, and if we do so, the result is a single phase circuit that can't be faster than what the original retiming ([LEI 91]) would give. Once the schedule in done, we place registers with an $O(|E|)$ algorithm without looking at where they were previously. Our method has as output a circuit with a multi-phase clock, neither the phases nor their count are fixed a priori like in [LOC 94] or [ISH 97]. Our method is not an approximation, and the running time is low even for circuits with many phases, unlike [ISH 97]. We do not handle the problem of finding a solution with constraints on the clock phases, which is done in [ISH 97] by having a fixed number of phases and permitting to do retiming with fixed duty ratios.

The main contributions of this paper are the following:

- We do not limit ourselves to edge-triggered flip-flops or level sensitive latches only, but our proposed solution can use a mixture of the two, which is automatically found by a linear algorithm.
- The overall complexity of our method is $O(|V|\,|E|\log(|V|\,d_{max}))$, or $O(|V|\,|E|\,d_{max})$ for small integral delays, where $|V|$ is the number of computing elements in the circuit, $|E|$ is the number of connections between these elements and d_{max} is the maximum duration of the computations done by the elements. The complexity of the retiming method is $O(|V|\,|E|\log|V|)$. Even if in the general case the overall complexity of the two approaches is similar, we avoid the computation of all pair-shortest paths, which is a practical burden regarding both space and time. Our method has an upper bound not higher than any clock minimization method described in previous work cited in this paper.
- The optimal solution to the clock period minimization problem is always achieved.
- Some combinational functions may have a delay greater than the clock period. In this case, the optimal throughput can be reached by increasing the number of functional units realizing the function.

This paper is organized as follows. Section 2 recalls the notations and definitions used in this work. Section 3 presents the main algorithm used as a replacement to the retiming approach. It gives also the main theorems concerning the validity of our approach. Section 4 gives the algorithms for register placement and the automatic selection of edge-triggered or level-sensitive storage. Section 5 extends the method to non-integer clock periods and combinational logic with delays greater than the clock period. Section 6 presents the implementation and experimental results. Section 7 concludes the paper and points to some future work.

2 Preliminaries

2.1 Input Circuit Definition

As in [LEI 91], the input circuit is formed by combinational computing elements separated by registers. We model that circuit as a finite, vertex-weighted, edge-weighted, directed multigraph $G = \langle V, E, d, w\rangle$. The vertices V represent the functional elements of the circuit, and they are interconnected by edges E. Each vertex $v \in V$ has a propagation delay $d(v) \in \mathbb{Q}$ which is the maximum delay before its outputs stabilize, and we do not suppose there is a minimum delay before its outputs change. Each edge $e \in E$ is weighted with a register count $w(e) \in \mathbb{N}$ representing the number of registers separating two functional elements. We extend the d and w functions for paths in the graph. For any path $p = v_0 \xrightarrow{e_0} v_1 \xrightarrow{e_1} \dots \xrightarrow{e_{k-1}} v_k$, we define:

$$d(p) = \sum_{i=0}^{k-1} d(v_i) \qquad\qquad w(p) = \sum_{i=0}^{k-1} w(e_i)$$

Note that, unlike [LEI 91], $d(p)$ does not take into account the weight of the last vertex. We also use the following notations: if v_i and v_j denote vertices then e_{ij} will designate the edge that goes from v_i to v_j . Given a specific path, e_i denotes the edge that goes from a vertex (v_i) to the next vertex (v_{i+1}) in the path.

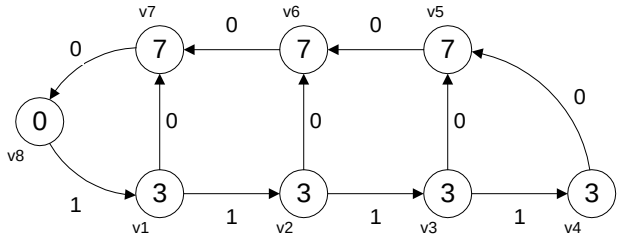


Figure 1: A simple correlator circuit.

Figure 1 shows the graph for the correlator example in [LEI 91]. This circuit does an iterative process: at each clock cycle, the circuit calculates new values from the previously calculated ones. The register counts can be thought of as the number of iterations between the time the value is calculated and the time it is used (e.g. in Figure 1 the element v2 uses the result of the previous iteration of element v1). By thinking of the graph as an inter and intra-iteration dependency graph, instead of number of registers, we can use an algorithm for optimal loop scheduling [DON 92] to have the maximal throughput, which is limited only by data dependencies and propagation delays. This schedule is not limited by the clock period or the position of the registers (proved in LEMMA 5, page 7). Register placement is performed at a subsequent step that takes the results of the scheduling step as input.

2.2 Retiming ([LEI 91])

Retiming can be viewed as a displacement assigned to each vertex, which affects the length (weights) of the edges. More formally, a retiming on an edge-weighted graph $\langle V, E, w\rangle$ is a function $r : V \to \mathbb{Z}$ (or $V \to \mathbb{Q}$ when the weights w can be fractional, which gives a more general graph transformation than in [LEI 91]) that transforms it in a new graph $\langle V, E, w_r\rangle$ where the weights w_r on the edges are defined as:

$$w_r(e_{ij}) = w(e_{ij}) + r(v_j) - r(v_i) \qquad [1]$$

A property of retiming:

$$\text{for any path } v_i \xrightarrow{p} v_j,\ w_r(p) = w(p) + r(v_j) - r(v_i) \qquad [2]$$

This implies that for any directed cycle c (a path from a vertex to itself), $w_r(c) = w(c)$.

2.3 Schedule

We define $s_n(v_i)$ to be the time at which the n^{th} iteration of operation v_i is starting. A schedule s is said periodic (all iterations having the same schedule), with a period of P, if:

$$\forall n,\ \forall v_i \in V,\ s_{n+1}(v_i) = s_n(v_i) + P$$

A schedule is said k-periodic if there exist integers n_0, k and a positive rational P such that:

$$\forall n \geq n_0,\ \forall v_i \in V,\ s_{n+k}(v_i) = s_n(v_i) + P\ k$$

Both periodic and k-periodic schedules have the same throughput $T = {}^1/_P$ (this is called the frequency in some papers but it causes a bit of confusion with the clock frequency), but the k-periodic schedule has a period of $P\ k$. A schedule is valid iff the operations terminate before their results are needed, while respecting resource constraints if any.

3 Scheduling operations

First we must find the critical cycle in the circuit; that is, the cycle that limits the throughput. The maximum throughput is [DON 92]:

$$T = \min_{c_k}\left\{\frac{w(c_k)}{d(c_k)}\right\}$$

where c_k is a directed cycle in the graph. If there are no cycles in the graph, T is infinite; this means that we can calculate all the values at the same time, if we have enough resources to do so. Computing the maximal throughput is a minimal cost-to-time ratio cycle problem, which can be solved in the general case in $O(|V||E|\log(|V|\,d_{max}))$. The method is based on iteratively applying the Bellman-Ford's algorithm for longest paths on a new graph $G_l = \langle V, E, w_l\rangle$ derived from G, with $w_l(e_{ij}) = d(v_i) - P\,w(e_{ij}) \in \mathbb{Q}$, where $P = {}^1/_T$ is the period, to find the minimal value of P for which there are no positive cycles in this graph [DON 92, BEN 95]. For small integral delays, we can compute the maximal throughput in $O(|V||E|\,d_{max})$ [HAR 91]. For the circuit of Figure 1, we find that the minimal period is $P = 10$, which is interesting compared to retiming that gave a minimal period of 13. This result is the same as what was published in [LOC 94], but the approach diverges from this point, since retiming is used on a modified graph in [LOC 94], compared to loop scheduling that we use in this work. Note that, as we mentioned previously, the method proposed in [LOC 94] cannot always find the optimal throughput if it is not given the right set of phases. Figure 2 shows the graph for $P = 10$, where the vertices are labeled by the length of their longest paths from/to v1.

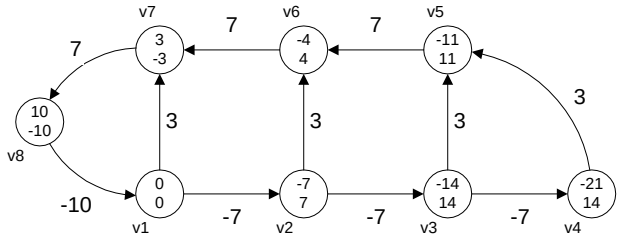


Figure 2: G_l with the longest paths from/to v1 in the vertices.

The weights denote the minimum distance between the schedule of two vertices. For example the -7 between v1 and v2 means that v1 must be scheduled at most 7 units of time after v2, the 3 between v1 and v7 means that v7 must be scheduled at least 3 units of time after v1, etc. Finding the longest paths in this graph gives a possible schedule with a period of P. This can be done in $O(|V||E|)$ with Bellman-Ford's algorithm. The ASAP and ALAP schedules can be obtained by finding the longest paths to and from a chosen vertex. To find the longest paths to a vertex, Bellman-Ford's algorithm can be applied on the graph $G_l^{-1} = \langle V, E^{-1}, w_l^{-1}\rangle$ derived from G_l, where $e_{ij} \in E^{-1} \Leftrightarrow e_{ji} \in E$ and $w_l^{-1}(e_{ij}) = w_l(e_{ji})$. Given a specific vertex v_1, the ASAP (*resp*. ALAP) schedule of any other vertex is the longest path (resp. minus the longest path) from v_1 to that vertex using the G_l (*resp.* G_l^{-1}) graph. The longest path from a vertex to itself gives us its mobility. The mobility can also be obtained as the difference between the ALAP and ASAP schedule times or vice-versa.

LEMMA 1: The longest path from v_i to v_j, two vertices in the graph G_l, is independent of the placement of the registers in the graph G (that is, for all possible retiming r of the graph), and $w_{r\,l}(p) = w_l(p) - P\,(r(v_j) - r(v_i))$.

PROOF: By [2] and because $d(p)$ does not change with retiming, we have $w_{r\,l}(p) - w_l(p) = -P\,(w_r(p) - w(p)) = -P\,(r(v_j) - r(v_i))$. This means that all paths from v_i to v_j have changed in length by the same value after retiming, so the longest path stays the same but with a different length $w_{r\,l}(p) = w_l(p) - P\,(r(v_j) - r(v_i))$. □

Lets define $l(i,j)$ to be the length of the longest path from v_i to v_j in G_l. Table 1 gives l for the graph in Figure 2.

Table 1: Longest paths in graph G_l, only the values in bold are calculated.

	1	2	3	4	5	6	7	8
1	**0**	**-7**	**-14**	**-21**	**-11**	**-4**	**3**	**10**
2	**7**	0	-7	-14	-4	3	10	17
3	**14**	7	0	-7	3	10	17	24
4	**14**	7	0	-7	3	10	17	24
5	**11**	4	-3	-10	0	7	14	21
6	**4**	-3	-10	-17	-7	0	7	14
7	**-3**	-10	-17	-24	-14	-7	0	7
8	**-10**	-17	-24	-31	-21	-14	-7	0

Table 2: Schedules and mobility relative to v_1.

vertex	1	2	3	4	5	6	7	8
ASAP	0	-7	-14	-21	-11	-4	3	10
ALAP	0	-7	-14	-14	-11	-4	3	10
Mobility	0	0	0	7	0	0	0	0
Interval	0	-7	-14	[-21,-14]	-11	-4	3	10

$l(i,j)$ gives the relative schedule of vertices for the same iteration, that is we have $s_n(v_j) - s_n(v_i) \geq l(i,j)$. This permits to determine intervals in which an operation must be scheduled relatively to another operation. Also, because we want a periodic schedule with a period of P, we have that:

$$l(i,j) + P\,m \leq s_{n+m}(v_j) - s_n(v_i) \leq -\,l(j,i) + P\,m \qquad [3]$$

For example looking at Table 1 we know that $s_n(v_2) - s_n(v_1) \geq -7$ and $s_n(v_1) - s_n(v_2) \geq 7$ which means that $s_n(v_2) - s_n(v_1) = -7$. Keeping only one line, and the corresponding column, for a vertex that is on the critical cycle, we find the intervals where we can schedule the vertices. This means that we do not need to compute all-pairs longest paths but only the longest path from and to that vertex. Table 2 presents the schedule intervals relative to vertex v_1.

We represent the periodic schedule of the operations (vertices), with a period of P, by a schedule graph $G_s = \langle V, E, d, w_s, P \rangle$. G_s is derived from the graph G and a schedule s, and the weights are $w_s(e_{ij}) = s_{w(e_{ij})}(v_j) - s_0(v_i) \in \mathbb{Q}$. The weights w_s represent the time distance between the start of an operation and the start of the operation that needs the result from this one. G_s is consistent iff it has the following properties:

1. For all edge e_{ij} $w_s(e_{ij}) \geq d(v_i)$
2. All paths p from v_i to v_j have the same $w_s(p)$ mod P, which must be 0 if $v_i = v_j$.

Figure 3 shows a consistent G_s for our example, using s as the ALAP schedule from Table 2.

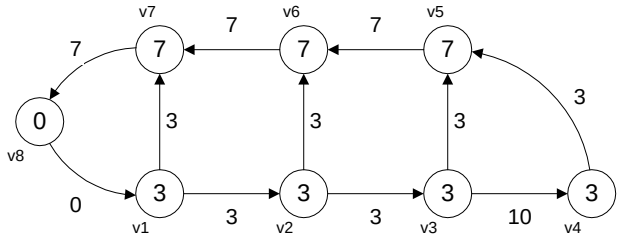


Figure 3: Schedule graph G_s.

LEMMA 2: the graph G_s is a retiming of the graph $\langle V, E, d, P\ w \rangle$, the retiming vector being the associated schedule.
PROOF: By definition of G_s we have that $w_s(e_{ij}) = s_{w(e_{ij})}(v_j) - s_0(v_i)$. G_s being made from a periodic schedule ($s_n = s_0 + P\ n$), this implies that $w_s(e_{ij}) = P\ w(e_{ij}) + s_0(v_j) - s_0(v_i)$. □

The graph with the edge-weights all multiplied by a constant c is called a *c-slow circuit*. The circuit has been slowed down by a factor of c, so that it does the same computation but it takes c times as many clock cycles ([LEI 91]). Therefore, the graph G_s could be interpreted as a circuit that does the same computations as G. A c-slow circuit can be retimed to have a shorter clock period but the throughput is not higher if we are doing only one computation at a time; multiple interleaved computations can be done with the circuit to have a better efficiency. This is not our interpretation of that graph and our final circuit is not c-slow, it produces results every clock cycle like the original circuit.

LEMMA 3: s being a periodic schedule, the graph G_s made from s is consistent iff s is valid.
PROOF: A schedule s is valid iff the operations terminate before their results are needed, which means that $s_n(v_i) + d(v_i) \leq s_{n+w(e_{ij})}(v_j)$. s being periodic, this is equivalent to $w_s(e_{ij}) \geq d(v_i)$ (first property of consistency) by definition of w_s. By LEMMA 2 and by a property of retiming, we have that for all paths from v_i to v_j, $w_s(p) \bmod P = (s_0(v_j) - s_0(v_i)) \bmod P$ (second property of consistency). □

LEMMA 4: Retiming a consistent graph G_s gives a consistent graph if the first property is kept (where the retiming values do not have to be integers).
PROOF: Direct from LEMMA 2 and LEMMA 3. □

A consequence of LEMMA 4 is that we can explore different schedules (all with the same period) by retiming the graph G_s to find one that is easier/smaller to implement.

LEMMA 5: The graph G_s is independent of the placement of the registers in the graph G (that is, for all possible retiming r of the graph).
PROOF: Let w_r, l_r and $w_{r\,s}$ be respectively the values of w, l and w_s after applying a retiming r to G. By [3] we have that $l(i,j) + P\ w(e_{ij}) \leq w_s(e_{ij}) \leq -\ l(j,i) + P\ w(e_{ij})$. By LEMMA 1, we have that $l_r(i,j) = l(i,j) - P\ (r(v_j) - r(v_i))$. This implies that:

$$l_r(i,j) + P\ w_r(e_{ij}) \leq w_{r\,s}(e_{ij}) \leq -\ l_r(j,i) + P\ w_r(e_{ij})$$

$$\Leftrightarrow\ l(i,j) - P\ (r(v_j) - r(v_i)) + P\ w_r(e_{ij}) \leq w_{r\,s}(e_{ij}) \leq -\ l(j,i) - P\ (r(v_j) - r(v_i)) + P\ w_r(e_{ij})$$

$\Leftrightarrow$ $l(i,j) + P\, w(e_{ij}) \le w_{r\,s}(e_{ij}) \le - l(j,i) + P\, w(e_{ij})$, by definition of $w_r(e_{ij})$.
$w_s(e_{ij})$ and $w_{r\,s}(e_{ij})$ being contained in the same interval, the graph G_s does not change when retiming G. □

4 Disposition of storage elements

4.1 Register placement

Register placement is derived from a schedule graph G_s . We suppose that for every vertex v_i, $d(v_i) \le P$. The general case will be dealt with in section 5. The easy way to place registers would be to place them before each operation and activate them according to the schedule but this would be a waste of space and it works only if $w_s(v_i) \le P$. Instead of registering every input of every function we propose to chain operations, and as a consequence we reduce the number of registers and controlling signals needed. In fact, we only need a register at each P time units. This is if we do not suppose that the output of combinational logic stays stable for some time after its inputs change. Therefore, we must break every path, in the graph, which is longer than P; we can put more than one register on an edge. We use a greedy algorithm, not necessarily optimal, for placing the registers but we do not claim that the placement is optimal. This algorithm is $O(|E|)$:

```
BreakPath( v_i, d, t )
 if vertex v_i already passed then return

 mark vertex v_i as passed
 v_i.distance := d

 for each edge e_ij out from this vertex do
   if v_j already passed and d + w_s(e_ij) > v_j.distance
        or d + w_s(e_ij) + max(w_s(e_jk)) > P then
     if w_s(e_ij) > P
       put ceiling(w_s(e_ij)/P)-1 registers on edge e_ij scheduled at time t
     put a register on edge e_ij scheduled at time (t + w_s(e_ij)) mod P
     BreakPath( v_j, 0, (t + w_s(e_ij)) mod P)
   else
     BreakPath( v_j, d + w_s(e_ij), (t + w_s(e_ij)) mod P)
```

Table 3: Register placement and schedule.

Name	*edge*	*schedule*
A	$2 \to 3$	6
B	$3 \to 4$	6
C	$5 \to 6$	6
D	$6 \to 7$	3
E	$8 \to 1$	0
F	$2 \to 6$	6

Table 3 gives the register placement and schedule starting `BreakPath` with (v_1, 0, 0). Figure 4 shows the placement of registers in the final circuit, according to Table 3. Figure 4b shows the

delay graph, where the vertices represent registers and the edges are labeled by the length of the path between two registers adjacent in G_s.

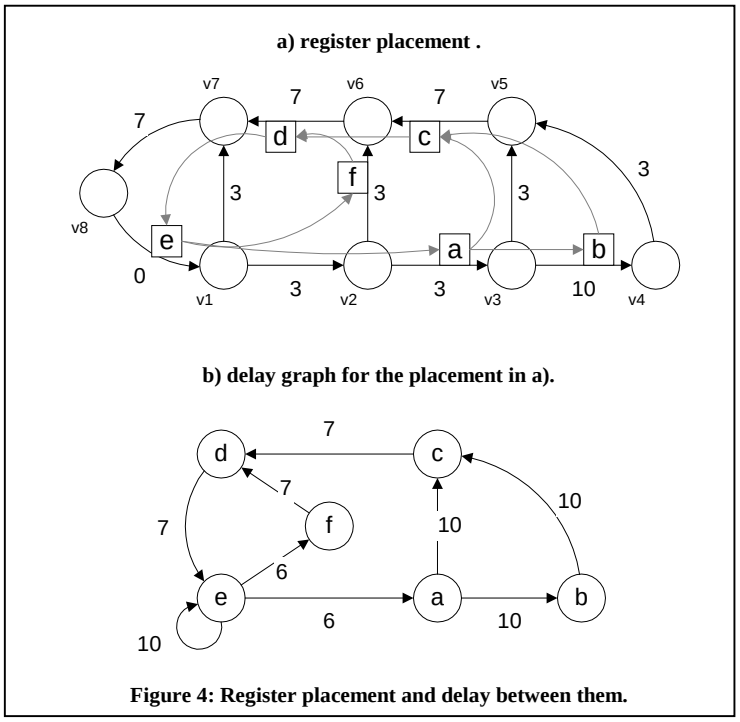


Figure 4: Register placement and delay between them.

4.2 Latch determination

From the delay graph, we can determine which register can be a latch and when we may enable and disable it. The idea is the same as above, there must be no path longer than *P*; that is, there must be no more than *P* units of time between the enable of a register and the disable of its successors. Also, a register must be enabled at the time it is scheduled: for example, register *a* must be enabled at time 6. This means that the maximum time a register can be enabled after (before) its scheduled time is *P* minus the maximum of the lengths of the edges that go to (exit from) that register. In our example, register *e* must be enabled exactly at time 0 because there is an edge of length 10 from and to *e*, so it will be edge-triggered, but register *f* can stay enabled 4 units of time after 6 and can be enabled 3 units of time before, so it can be level-sensitive with enable at time 3 and disable at time $10 \equiv 0 \bmod P$. A possible way to enable registers is:

name	***enable time***
a	[6, 0[
b	6
c	6
d	[0,6[
e	0
f	[6, 0[

This can be done with a two-phase clock, *e* and *d* being clocked by the first phase and *a*, *b*, *c* and *f* being clocked by the second one. *b*, *c* and *e* are edge-triggered and *a*, *d* and *f* are level-sensitive. This solution has the same period as the one in [LOC 94], but assuming that the cost of an edge-triggered register is *R* and a latch *R*/2, the storage element cost for their circuit is 5*R* while ours is

4.5*R*, this represents a 10% improvement in the area occupied by the registers. Also, we can use a two-phase clock with an underlap between phases without changing the period.

The algorithm `PlaceLatches` checks each register in the delay graph to see if its flip-flops can be replaced by latches and it gives the enable and disable time for each latch. Each vertex contains its schedule time, which has been given by `BreakPath`. This algorithm is $O(|E|)$. It is an efficient but not necessarily optimal way to place latches and changing the order in which the vertices are processed may give better results.

```
PlaceLatches:
  for each vertex v_i do
    after = P - max_j{w(e_ji)}
    before = P - max_j{w(e_ij)}
    if after ≠ 0 or before ≠ 0 then
      set v_i as a latch
      set enable time to be (scheduled time - before) mod P
      set disable time to be (scheduled time + after) mod P
      set w(e_ji) to be w(e_ji) + after for each edge e_ji
      set w(e_ij) to be w(e_ij) + before for each edge e_ij
```

Proof that the circuit with latches will store valid values in its registers, and thus will have a correct comportment:
From a delay graph with a period of P, let suppose that there are vertices from registers $A_1, A_2, \ldots, A_n$ to the register B, and that these vertices have delays of $d_1, \ldots, d_n$ respectively. Register A_i is enabled on the interval $[a_i, b_i[$ and was originally scheduled at time t_i. Register B is enabled on the interval $[a_B, b_B[$ and was originally scheduled at time t_B. An edge-triggered register is modeled with $a_i = b_i$, which means that there is no time where the value passes through the register but we still consider that the input value just before time a_i is stored at time a_i.

We want to prove that if for all i, A_i is valid and stable on time intervals $[t_i, a_i + P[$, then B will be valid and stable on time interval $[t_B, a_B + P[$.

By LEMMA 1 we have that $t_i + d_i \leq t_B$, which implies that all inputs of B are valid at time t_B if A_i are valid and stable on time interval $[t_i, t_B[$. By definition we have that $a_i \leq t_i \leq b_i$ and $a_B \leq t_B \leq b_B$. Also, because there are no path longer than P, $b_B \leq a_i + P$. On time interval $[t_B, b_B[$, B is enabled and its inputs are valid and stable because A_i is valid and stable on time interval $[t_i, a_i + P[$ and that $t_B \leq b_B \leq a_i + P$. On time interval $[t_B, a_B + P[$, B is disabled so that it stays stable and was valid at the disable time, which means that it is still valid. So, register B will be valid and stable on time interval $[t_B, a_B + P[$. □

5 Extensions

5.1 Functional elements of duration greater than the clock period

Suppose we want to do a scalar product. We can do this using a simple multiply and accumulate circuit as shown in Figure 5 (additions taking one time unit and multiplications taking two). We should notice that the multiplication (v_1) is longer than the period (P). If $d(v_i) > P$, we have that $s_n(v_i) + d(v_i) > s_{n+1}(v_i)$, which means that we must start the next calculation in vertex v_i before the current one finishes. There are two ways to accomplish this: pipeline v_i (Figure 5b) or put multiple instances of v_i (Figure 5c). To be able to pipeline v_i, we must ask the designer (or a synthesis tool) to split the calculations in v_i so that each part has a delay $\leq P$.

To put multiple instances of v_i, we can modify `BreakPath` so that when $d(v_i) > P$ it places $\left\lceil \frac{d(v_i)}{P} \right\rceil$ functional elements like v_i, each one with registered inputs and multiplexing the outputs.

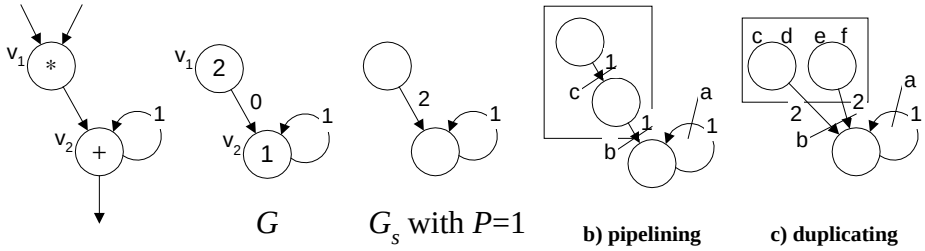


Figure 5: Scalar product example.

At each period, the input registers from the next unit are enabled and the multiplexor chooses the right unit for output. We must adjust $d(v_i)$ to include the delay of the multiplexor and find a new valid schedule and then restart `BreakPath`. In our example, the input registers and output are scheduled as shown in Table 4.

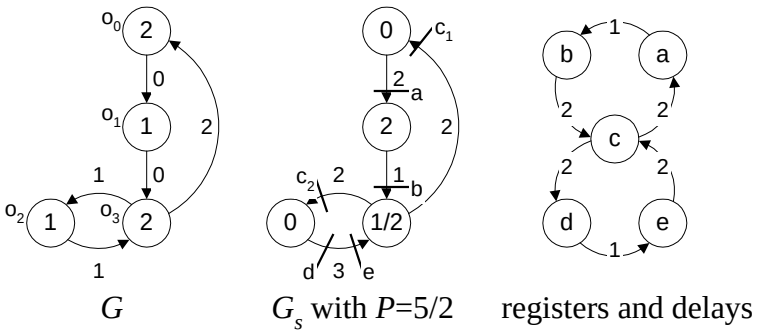
Table 4: Schedule of duplicated unit in the scalar product example.

Cycle	*enable reg.*	*Output*
Even	c, d	e*f
Odd	e, f	c*d

5.2 Fractional clock duration and k-periodic scheduling

Suppose we want to find an optimal circuit for the circuit graph G in Figure 6. Figure 6 shows a possible schedule and the register placement done by `BreakPath`. We should notice that P is fractional. Also, registers c_1 and c_2 are in fact only one register because they have the same input and they have the same schedule. This 1st solution may be acceptable but it must have a clock resolution smaller than the time unit used. If all the $d(v_i)$ are integers then there exist an optimal valid schedule with $s_n(v_i)$ being all integers. In fact, if all the $d(v_i)$ are integers, a theorem reported in [BEN 95, DON 92] says that if we have a valid periodic schedule s then the schedule s^* defined as $s^*_n(v_i) = \lfloor s_n(v_i) \rfloor$ is a valid k-periodic schedule with the same throughput.

Applying the theorem on the fractional schedule we used in the 1st solution gives the new schedule in Table 5. The period of this circuit is 5 but in each period two output values are calculated so we have the same throughput as in the 1st solution.



name	*enable time*
a	$[\frac{1}{2}, 2[$
b	$\frac{1}{2}$
c	$[2, \frac{1}{2}[$
d	$[\frac{1}{2}, 2[$
e	$\frac{1}{2}$

Figure 6: k-periodic example, 1st solution

Table 5: filter example, 2nd solution; k-periodic schedule with *k*=2 and *P*=5/2

name	*schedule₁*	*schedule₂*	*enable times*
a	2	4	{ [3,2[}
b	3	0	{ 3, 0 }
c	0	2	{ 0, 2 }
d	2	4	{ [3,2[}
e	3	0	{ 3, 0 }

6 Experimentation and implementation

For our experimentation, we are using a tool that we developed primarily for loop acceleration *called L.A.* It accepts a description in standard C and produces an internal format where cyclic behaviors are explicit. This intermediate format can be used as input to different algorithms and CAD tools that we intend to develop in the future. To facilitate the development we started from a retargetable C Compiler meant to be modified and retargeted easily [FRA 95]. The first benchmark is the one presented in [LEI91, LOC94]. In order to compare our results we also implemented the retiming method presented in [LEI91]. The acceleration is zero for examples where only one clock phase is needed to have an optimal schedule, but varies from 9% to 100% when more pipelining is possible with multiple phases.

We developed the scalar product example and translated from VHDL to C some examples from the HLSynth92 benchmark suite [HLB 96]. It is interesting to note that the filter specification in the suite cannot be accelerated but by re-writing the specification, using tree balancing of the expressions, we obtain an acceleration of 150% using retiming and an additional 9% using our method.

	period		registers*		num. of phases	acceleration
	retiming	L.A.	retiming	L.A.		
correlator	13	10	5	4.5	2	30%
scalar product example	2	1	2	4	2	100%
k-periodic example	3	$^5/_2$	4	3.5 or 4	2 or 3	20%
diffeq	6	6	7	7	1	0%
ellipf	10	10	13	13	1	0%
modified ellipf	4	$^{11}/_3$	50	25.5	7	9%

* register count is the number of edge-triggered plus $^1/_2$ the number of level-sensitive storage.

7 Conclusion and future work

In this work, we showed that software pipelining techniques are an excellent alternative to retiming techniques in sequential circuit optimization. The resulting circuit has an optimal throughput using multi-phase-clocked circuits with a combination of edge-triggered and level sensitive storage. The computing complexity is similar to previously published methods but we have a guarantee of always obtaining the optimal solution regarding the throughput, according to the precision of the graph representation of the circuit. The phases are automatically computed and the registers placed by a greedy algorithm. Future work includes the design of an optimal algorithm to minimize the number of clock phases and the number of registers and maximize chaining. Benchmarks have shown that rewriting of the initial specification using algebraic transformations (like associativity and commutativity) can have a tremendous impact on the final result, we intend to augment our tool using such capabilities. Our work has to be extended to take into account clock skews and to minimize the impact of such phenomena on the overall performances. In addition, the circuit graph could have minimum delays on its edges, which is the time before the output of combinational logic start to change when the inputs are changed. This would permit to have paths longer than P between registers, which could reduce the number of registers. Tradeoffs between the number of phases, space and throughput have to be explored.

References

[BEN 95] I. E. Bennour and E. M. Aboulhamid, “Les problèmes d'ordonnancement cycliques dans la synthèse des systèmes numériques,” Université de Montréal, Montreal, Publication 996, Oct. 1995; http://www.iro.umontreal.ca/~aboulham/pipeline.pdf

[DON 92] H. V. Dongen, G. R. Gao, Q. Ning, “A Polynomial Time Method for Optimal Software Pipelining”, Parallel processing, CONPAR VAPPV 92, *Lectures Notes in Computer Sciences*, Vol. 634, 1992.

[DEO 95] R. B. Deokar, S. Sapatnekar, “A Fresh Look at Retiming via Clock Skew Optimization”, *DAC 95*.

[FRA 95] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*: Bunjamin Cummings, 1995.

[HAN 94] C. Hanen, “Study of NP-hard Cyclic Scheduling Problem: the Recurrent Job-Shop”, *European Journal of Operation Research*, Vol. 72, 1994.

[HAR 91] M. Hartmann, J. Orlin, “Finding minimum cost to time ratio cycles with small integral transit times”, *Networks; an international journal*, Vol. 23, 1993, pp. 567-574.

[HLB 96] http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html

[HWA 91] C.-T. Hwang, Y.-C. Hsu, Y.-L. Lin, “Scheduling for Functional Pipelining and Loop Winding”, *In proceedings of the Design Automation Conference* 1991.

[ISH 97] A. T. Ishii, C. E. Leiserson, M. C. Papaefthymiou, “Optimizing Two-Phase, Level-Clocked Circuitry”, *Journal of the ACM*, Vol. 44, No. 1, January 1997, pp. 148-199.

[LAW 76] E. Lawler, *Combinatorial Optimization: Networks and Matroids*: Saunders College Publishing, 1976.

[LEG 97] C. Legl, P. Vanbekbergen, A. Wang, “Retiming of Edge-Triggred Circuits with Multiple Clocks and Load Enables”, IWLS’97.

[LEI 91] C. E. Leiserson, J. B. Saxe, “Retiming Synchronous Circuitry”, *Algorithmica*, Vol. 6, 1991.

[LOC 94] B. Lockyear, C. Ebeling, “Optimal Retiming of Level-Clocked Circuits Using Symmetric Clock Schedules”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 13, No. 9, September 1994.

[MAH 97] N. Maheshwari, S. Sapatnekar, “An Improved Algorithm for Minimum-Area Retiming”, *DAC 97*, June 1997.

[MAH 98] N. Maheshwari, S. Sapatnekar, “Efficient Retiming of Large Circuits”, *IEEE Transactions on VLSI Systems*, Vol. 6, No. 1, pp. 74-83, March 1998.

[MIC 94] G. D. Micheli, Synthesis and Optimization of Digital Circuits: McGraw-Hill, Inc., 1994.