Optimal design of synchronous circuits using software pipelining techniques

François R. Boyer
and
El Mostapha Aboulhamid
Université de Montréal
and
Yvon Savaria
École Polytechnique de Montréal
and
Michel Boyer
Université de Montréal

We present a method to optimize clocked circuits by relocating and changing the time of activation of registers to maximize the throughput. Our method is based on a modulo scheduling algorithm for software pipelining, instead of retiming. It optimizes the circuit without the constraint on the clock phases that retiming has, which permits to always achieve the optimal clock period. The two methods have the same overall time complexity, but we avoid the computation of all pair-shortest paths, which is a heavy burden regarding both space and time. From the optimal schedule found, registers are placed in the circuit without looking at where the original registers were. The resulting circuit is a multi-phase clocked circuit, where all the clocks have the same period and the phases are automatically determined by the algorithm. 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.

Categories and Subject Descriptors: B.6.1 [**Logic Design**]: Design Styles—*Sequential circuits*; B.6.3 [**Logic Design**]: Design Aids—*Optimization/Automatic synthesis*

General Terms: Performance, Algorithms

Additional Key Words and Phrases: retiming, software pipelining, resynthesis

A shorter version (6 pages) of this paper appeared in the Proceedings of ICCD '98, pp. 62-67.
Name: François R. Boyer
Affiliation: DIRO, Université de Montréal
Address: 2920 Ch. de la Tour, C.P. 6128, Succ. Centre-Ville, Montréal, (Qué), Canada, H3C 3J7
E-mail: boyerf@IRO.UMontreal.CA

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. For circuits synchronized by a single periodic event (single clock), the wasted time between two storage elements (registers) is the period duration minus the combinational delay between these storage elements. In this paper we focus on methods for minimizing that wasted time. The proposed method also identifies the circuit path that limits the speed, for further optimization if necessary.

Leiserson and Saxe [1991] 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 period. 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 in many cases that this solution is 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; better results could be obtained if some form of fractional retiming was applied.

Lockyear and Ebeling [1994] 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 reduce wasted time. That method will give an optimal solution, with no wasted time on the critical path, but only if the phases specified allow it and under the assumption of using a perfect clock.

Deokar and Sapatnekar [1995] 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 that equivalence to retime the circuit to bring down the "skews" to zero (or as close as possible). If the "skews" are not zero, they can be forced to zero (increasing slightly the clock period) or circuitry can be added to implement 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 [1997, 1998] use retiming to reduce the area (number of registers) 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 linear programming problem to be solved.

Ishii, Leiserson, and Papaefthymiou [1997] present methods to minimize the clock period on a multi-phase level sensitive clocked circuit. They also show how to convert an edge-triggered clocked circuit into a faster level-sensitive 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 $O(k\,|V|^3)$, where $|V|$ is the number of computing elements in the circuit. 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 of a two-phase circuit, an approximation, with a period at most $(1+\epsilon)$ times the optimal, can be found in $O\left(|V|^3 \frac{1}{\epsilon} \log \frac{1}{\epsilon} + \left(|V||E| + |V|^2 \log |V|\right) \log \frac{|V|}{\epsilon}\right)$, where $|E|$ is the number of connections between elements. For k-phase, the running time for that approximation contains a factor of ϵ^{-k}, which is impractically large for small values of ϵ.

An optimal solution to the clock tuning problem, on multi-phases level-sensitive circuits, with fixed register placement, is presented by Sakallah, Mudge and Olukotun [1990]. The algorithm uses linear-programming, but no running time or upper bound is given.

Legl, Vanbekbergen, and Wang [1997] extend retiming to handle circuits where not all the registers are enabled at the same time. The idea is that registers can be moved 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 Van Dongen, Gao, and Ning [1992] 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, some methods use retiming as a heuristic to find schedules under resource constraints [Bennour and Aboulhamid 1995].

We present a method that uses 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 [Deokar and Sapatnekar 1995; Maheshwari and Sapatnekar 1997; Maheshwari and Sapatnekar 1998], but then we do not apply retiming according to that schedule. As said previously, not taking into account the short path problem can cause unpredictable circuit behavior when the skews are not forced to be zero. However, reducing the skews to zero results in a single-phase circuit, the same limitation as the original retiming [Leiserson and Saxe 1991]; we are considering the worst case length for short paths. Once the schedule is done, we place registers with an $O(|E|)$ algorithm, independently of their original placement. Our method produces a circuit with a multi-phase clock; neither the phases nor their count is fixed a priori like in some previous work [Ishii et al. 1997; Lockyear and Ebeling 1994]. Our method is not an approximation, and the running time is low even for circuits with many phases, unlike the work of Ishii et al. [1997]. We do not handle the problem of finding a solution with constraints on the clock phases, which is done in Ishii's work 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:

—The method is not limited 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 [Hartmann and Orlin 1993] 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 $O(|V||E|\log|V|)$ [Leiserson and Saxe 1991]. Even if the overall complexity of the two approaches is similar, we avoid the computation of all pair-shortest paths, which is a heavy 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 introduces 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

In this section, we define the graph representation of a sequential circuit, the "retiming" transformation of an edge-weighted graph, the short and long path constraints, and the basic notion of schedule.

2.1 Input Circuit Definition

As in the original retiming article [Leiserson and Saxe 1991] 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[1] $G = \langle V, E, d, w\rangle$. The vertices V represent the functional elements of the circuit, and they are interconnected by edges E. Each vertex $v \in V$ has a non-negative rational propagation delay $d(v) \in \mathbb{Q}$ which is the maximum delay before its outputs stabilize; no minimum delay is assumed (we use zero as lower bound on the short path, which is a common conservative approach). 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 functions d and w to paths in the graph. For any path $v_0 \overset{p}{\rightsquigarrow} v_k = v_0 \overset{e_0}{\to} v_1 \overset{e_1}{\to} \dots \overset{e_{k-1}}{\to} v_k$ we define

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

Note that, unlike Leiserson and Saxe's definition of $d(p)$ [1991], $d^-(p)$ does not take into account the weight of the last vertex ($d(p) = d^-(p) + d(v_k)$).

[1]Strictly speaking, G should also include the two functions $h : E \to V$ (head of edge) and $t : E \to V$ (tail of edge) such that if $v \overset{e}{\to} v'$ then $h(e) = v'$ and $t(e) = v$. There may be many edges e with same head and tail.

Fig. 1 shows the graph for the correlator example of Leiserson and Saxe [1991].

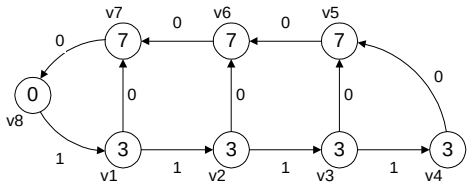


Fig. 1. A simple correlator circuit.

This circuit performs an iterative task: at each clock cycle, the circuit calculates new values from the previously calculated ones. The register count $w(e)$ for $v \xrightarrow{e} v'$ can be thought of as the number of iterations between the time the value is calculated by v and the time it is used by v' (e.g. in Fig. 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 [Van Dongen et al. 1992] 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). Register placement is performed at a subsequent step that takes the results of the scheduling step as input.

2.2 Retiming

Retiming [Leiserson and Saxe 1991] can be viewed as a displacement assigned to each vertex, which affects the length (weights) of the edges, taking weight from one side and putting it on the other. More formally, a *retiming* on an edge-weighted graph $\langle V, E, w\rangle$ is a function $r : V \longrightarrow \mathbb{Z}$ (or $V \longrightarrow \mathbb{Q}$ when edge weights $w(e)$ can be fractional, which gives a more general graph transformation) that transforms it into a new *retimed* graph $G_r = \langle V, E, w_r\rangle$, where the weights $w_r(e)$ for $v \xrightarrow{e} v'$ are defined by:

$$w_r(e) = w(e) + r(v') - r(v)$$

which directly implies that, for any path $v \overset{p}{\rightsquigarrow} v'$,

$$w_r(p) = w(p) + r(v') - r(v) \tag{1}$$

2.3 Short and long paths

When you change the inputs of a circuit, after some time the outputs start to change and may oscillate before stabilizing to the appropriate result. The short path is the least time the circuit takes before the output is affected by some input variation. Similarly, the long path is the longest time possible between the inputs change and the outputs stabilize. When designing a sequential circuit, we must have bounds on the short and long paths to know when the result is valid and when registers can be activated. These times are always positive (the outputs change after the inputs), so we can use zero as a lower bound. Using some tighter bounds could

permit more optimizations. The delays in our circuit graph are the upper bounds on the long paths.

2.4 Scheduling and software-pipelining

A *schedule* s [Bennour and Aboulhamid 1995; Van Dongen et al. 1992; Hanen 1994; Hwang et al. 1991; De Micheli 1994] is a function $s : \mathbb{N} \times V \longrightarrow \mathbb{Q}$, where $s_n(v) \equiv s(n, v)$ denotes the time at which the n^{th} iteration of operation v is starting. A schedule s is said to be *periodic* with period P (all iterations having the same schedule), if:

$$\forall n \in \mathbb{N}, \forall v \in V \; s_{n+1}(v) = s_n(v) + P$$

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

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

Both periodic and k-periodic schedules have the same throughput $\omega = {}^1\!/_P$ (also called "frequency" in some papers, causing some confusion with the clock frequency), but k-periodic schedules have a period of Pk. A schedule is said to be valid iff the operations terminate before their results are needed (whilst respecting resource constraints if any). If the only constraints come from data dependency, s is *valid* iff for all edges $v \xrightarrow{e} v'$,

$$s_n(v) + d(v) \leq s_{n+w(e)}(v').$$

3. SCHEDULING OPERATIONS

In this section, we show how to find the theoretical maximum throughput of a circuit (due to data dependency), and then, how to make a schedule that has a specified throughput. The scheduling is based on a loop-acceleration technique used in software pipelining.

3.1 Maximum throughput

First we must find the critical cycle in the circuit, i.e. the cycle $v \overset{c}{\rightsquigarrow} v$ that limits the throughput. The maximum throughput is [Van Dongen et al. 1992]:

$$\omega = \min_{c \in \mathcal{C}} \left\{ \frac{w(c)}{d^-(c)} \right\}$$

where $\mathcal{C}$ is the set of directed cycles in G.

If there is no cycle (if $\mathcal{C} = \varnothing$), then ω is infinite; this means that we can compute all the iterations 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 [Lawler 1976], which can be solved in the general case in $O(|V||E| \log(|V| d_{\max}))$ where $d_{\max} = \max_{v \in V} d(v)$. The method is based on iteratively applying Bellman-Ford's algorithm for longest paths on the graph $G_l = \langle V, E, w_l \rangle$ derived from G by letting

$$w_l(e) = d(v) - Pw(e) \in \mathbb{Q}$$

where $v \xrightarrow{e} \cdot \in E$ and $P = {}^1\!/_\omega$ is the period. A binary search is used to find the minimal value of P for which there is no positive cycle in G_l [Bennour and

Aboulhamid 1995; Van Dongen et al. 1992]. For small integral delays, we can compute the maximal throughput in $O(|V||E|d_{\max})$ [Hartmann and Orlin 1993]. For the circuit of Fig. 1, the minimal period P is equal to 10, which is interesting compared to retiming [Leiserson and Saxe 1991] that gave a minimal period of 13. This value that we obtain using loop scheduling is the same as that Lockyear and Ebeling [1994] obtained using retiming on a modified graph; their approach, though, may fail to give the optimal throughput if it is not given the right set of phases.

3.2 Schedule of a specified throughput

The graph G_l (whose weight function w_l is described above) is used to find a valid schedule with the specified throughput. Fig. 2 shows the graph for $\omega = 1/10$ ($P = 10$), where the vertices are labeled by the length of their longest paths from/to v1.

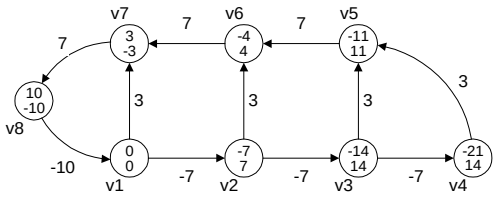


Fig. 2. G_l with the longest paths from/to v1 in the vertices.

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. A cycle with a positive length gives constraints that cannot be satisfied; the graph will have no positive cycle iff the period is feasible. The longest paths can be found in time $O(|V||E|)$ 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 transposed graph $G_l^\top$ of G_l where all the edges in G_l are made to point in the opposite direction, i.e. what was the head of an edge becomes its tail and conversely. Given a specific vertex v, the ASAP (resp. ALAP) schedule of any other vertex v' is the longest path (resp. minus the longest path) from v to v' in G_l (resp. $G_l^\top$). 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. Let $l(v, v')$ be the length of the longest path from v to v' in G_l. Table 1 gives l for the graph in Fig. 2.

The length $l(v, v')$ gives the relative schedule of vertices for the same iteration, that is we have $s_n(v') - s_n(v) \geq l(v, v')$. 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:

$$l(v, v') + Pm \leq s_{n+m}(v') - s_n(v) \leq -l(v', v) + Pm \tag{2}$$

	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 1. Longest paths in graph G_l; only the values in bold are calculated.

For example, looking at Table 1 we know that $s_n(\mathsf{v2}) - s_n(\mathsf{v1}) \geq -7$ and $s_n(\mathsf{v1}) - s_n(\mathsf{v2}) \geq 7$ which means that $s_n(\mathsf{v2}) - s_n(\mathsf{v1}) = -7$. 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 v1.

3.3 Schedule (multi)graphs

We represent a periodic schedule of operations (vertices), with period P, by a *schedule graph* $G_s = \langle V, E, d, w_s, P\rangle$, where d is as usual a delay function on vertices, and $w_s : E \to \mathbb{Q}$ is a weight function which associates to each $v \xrightarrow{e} v'$ the time distance between the start of operation v and that of operation v' using v's output as input. Schedule graphs are required to satisfy the following condition:

$$\text{If } v \overset{p}{\rightsquigarrow} v' \text{ and } v \overset{p'}{\rightsquigarrow} v' \text{ then } w_s(p) - w_s(p') = P\,k \text{ for some integer } k.$$

i.e. all paths between two fixed vertices have the same length modulo the (possibly fractional) period. To every multigraph $G = \langle V, E, d, w\rangle$ specifying a synchronous circuit and every P periodic schedule s on G (valid or not), there is an associated schedule graph $G_s = \langle V, E, d, w_s, P\rangle$ where $w_s(e)$ for $v \xrightarrow{e} v'$ is defined by

$$w_s(e) = s_{w(e)}(v') - s_0(v)$$

Indeed, $w_s(e) = s_0(v') - s_0(v) + Pw(e)$; if $v \overset{p}{\rightsquigarrow} v'$ and $v \overset{p'}{\rightsquigarrow} v'$, then $w_s(p) - w_s(p') = P\,k$ for $k = w(p) - w(p')$.

Definition 1. The graph G_s is *consistent* iff for all $v \xrightarrow{e} \cdot \in E$, $w_s(e) \geq d(v)$

Fig. 3 shows a consistent G_s for our example, using the ALAP schedule from Table 2 as s. The following lemma shows that the longest path from v to v' in G_l is independent of the placement of the registers in G.

	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

Table 2. Schedules and mobility relative to v1.

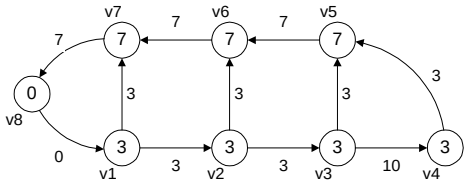


Fig. 3. Schedule graph G_s with $P = 10$.

LEMMA 1. *For all possible retimings r of G, p is a path of maximum length between v and v' in G iff it has the same property in the retimed graph G_r. Moreover $l_r(v, v') = l(v, v') - [r(v') - r(v)]P$, where $l_r(v, v')$ is the maximum length of a path between v and v' in $(G_r)_l$.*

PROOF. From $w_l(e) = d(v) - w(e)P$ for $v \xrightarrow{e} \cdot$, we deduce that for any path $v \overset{p}{\rightsquigarrow} v'$, $w_l(p) = d^-(p) - w(p)P$; moreover, if r is a retiming of G, then $w_r(p) = w(p)+r(v')-r(v)$ and so $w_{rl}(p) = d^-(p)-w_r(p)P = d^-(p)-[w(p)+r(v')-r(v)]P = w_l(p) - [r(v') - r(v)]P$. The result follows from the definitions of $l_r(v, v')$ and $l(v, v')$. □

LEMMA 2. *The graph G_s is the retimed graph derived from $\langle V, E, d, Pw\rangle$ where the retiming function $r(v)$ is $s_0(v)$ i.e. the schedule of iteration 0.*

PROOF. By definition of G_s, $w_s(e) = s_{w(e)}(v') - s_0(v)$ for $v \xrightarrow{e} v'$. Since s is periodic, $s_{w(e)}(v') = s_0(v') + w(e)P$ and so $w_s(e) = Pw(e) + s_0(v') - s_0(v)$. □

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 [Leiserson and Saxe 1991]. 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 improve the 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. *For all periodic schedules s, G_s is consistent iff s is valid.*

PROOF. A schedule s is valid iff the operations terminate before their results are needed, i.e. for all $v \xrightarrow{e} v'$, $s_n(v) + d(v) \leq s_{n+w(e)}(v')$; since s is periodic, this is equivalent to $s_0(v) + d(v) \leq s_{w(e)}(v')$ i.e. $w_s(e) \geq d(v)$ by definition of w_s. □

Notice that retiming a schedule graph always gives a schedule graph. Consequently:

LEMMA 4. *A retiming r of a consistent schedule graph G_s is legal iff it preserves consistency (i.e. $(G_s)_r$ is consistent).*

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.

The following result shows that retiming G has no influence on the schedule graphs provided we adjust the schedules accordingly.

LEMMA 5. *For any valid periodic schedule s and any legal retiming r of G, $s'_n(v) = s_n(v) - r(v)P$ is a valid periodic schedule of G_r such that $(G_r)_{s'} = G_s$.*

PROOF. If s is valid then for any $v \xrightarrow{e} v'$, $s_0(v) + l(v, v') \leq s_0(v')$. By LEMMA 1 $l_r(v, v') = l(v, v') - [r(v') - r(v)]P$ and so $s_0(v) - r(v)P + l_r(v, v') \leq s_0(v') - r(v')P$ i.e. $s'_n(v) = s_n(v) - r(v)P$ is a valid schedule of G_r. Moreover, $w_{rs'}(e) = w_r(e)P + s'_0(v') - s'_0(v) = w(e) + r(v') - r(v) + s_0(v') - r(v') - s_0(v) + r(v) = w_s(e)$ □

4. PLACEMENT OF STORAGE ELEMENTS

In this section, we show how the register types (edge-triggered or level-sensitive) and placements are obtained from the schedule graph in order to produce a circuit with the right functionality.

4.1 Register placement

Register placement is derived from a schedule graph G_s. In this section it will be assumed that for every vertex $v \in V$, $d(v) \leq P$. This assumption will be removed in section 5. The easy way to place registers is to place them before each operation and activate them according to the given schedule but this results in a waste of space and works only if $w_s(e) \leq P$. Instead of registering every input of every function we shall chain operations and, as a consequence, reduce the number of registers and controlling signals needed. In fact, we only need a register at each P units of time, considering that in the worst case a short path could be of length zero. 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 called `BreakPath` (Fig. 4), not necessarily optimal, for placing the registers. This algorithm is $O(|E|)$:

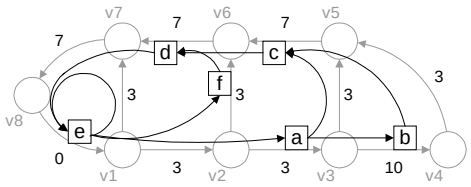
```
v.max_out ≡ max{w(e) : v --e--> ·}
proc BreakPath(v, dist, tim) ≡
  if (v.visited) then exit fi;
  v.visited = true;
  v.distance = dist;
  foreach (v --e--> v') do
    if ((v'.visited ∧ (dist + w_s(e) > v'.distance))
      ∨ (dist + w_s(e) + v'.max_out > P))
      then do
        if (w_s(e) > P)
          then put ⌈w_s(e)/P − 1⌉ registers on edge e
               scheduled at time tim; fi
        put one register on edge e
               scheduled at time (tim + w_s(e)) mod P;
        BreakPath(v', 0, (tim + w_s(e)) mod P); od
    else BreakPath(v', dist + w_s(e), (tim + w_s(e)) mod P)
  fi od.
```

Fig. 4. Algorithm to place registers

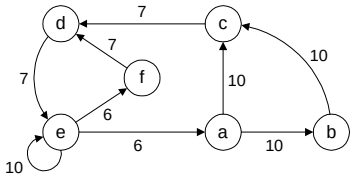
Table 3 gives the register placement and schedule starting `BreakPath` with $(\mathtt{v1}, 0, 0)$. Fig. 5 shows the placement of registers in the final circuit, according to table 3.

Name	*edge*	*Schedule*	*enable time*
a	1 → 3	6	[6, 0[
b	3 → 4	6	6
c	5 → 6	6	6
d	6 → 7	3	[0, 6[
e	8 → 1	0	0
f	2 → 6	6	[6, 0[

Table 3. Register placement and schedule.



a) register placement.



b) delay graph for the placement in a).

Fig. 5. Register placement and delay between them.

4.2 Latch selection

The selection of latch-type registers and their activation time are derived from the delay graph. 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 (which is equivalent to the phase non-overlap constraint presented by Sakallah et al. [1990]). 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, enabled at time 3 and disabled at time $10 \equiv 0 \bmod P$. Table 3 gives a feasible solution for registers' activation time, the intervals being the enable period of the latches (single values are for edge-triggered registers). The implementation of the circuit 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 presented by Lockyear and Ebeling [1994], but assuming that the cost of an edge-triggered register is R and that of a latch

is $R/2$, the storage element cost for their circuit is $5R$ whilst ours is $4.5R$. 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` (Fig. 6) 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.

```
PlaceLatches:
  foreach (v ∈ V) do
    v.max_out = max{w(e) : v --e--> ·}
    v.max_in = max{w(e) : · --e--> v} od
  foreach (v ∈ V) do
    after = P − v.max_in
    before = P − v.max_out
    if ((after ≠ 0) ∨ (before ≠ 0)) then do
      set v as a latch
      v.enable_time = (v.scheduled_time − before) mod P
      v.disable_time = (v.scheduled_time + after) mod P
      foreach (v' --e--> v) v'.max_out = max{v'.max_out, w(e) += after}
      foreach (v --e--> v') v'.max_in = max{v'.max_in, w(e) += before} od
    fi od
```

Fig. 6. Algorithm to place latches.

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.

LEMMA 6. *The circuit resulting from the algorithm PlaceLatches stores valid values in its registers, and will thus have a correct behaviour.*

PROOF. In a delay graph with period P, let B be any register and $A_1, \ldots, A_n$ be all those registers with a vertex to B and let $d_1, \ldots, d_n$ be their respective delays. 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 $[t_B, a_B + P[$. This will prove that the latches store valid values if we start with valid values.

By definition, we have that $a_i \leq t_i \leq b_i$ and $a_B \leq t_B \leq b_B$. Also, since there is no path longer than P, $b_B \leq a_i + P$. The validity of the schedule guarantees that $t_i + d_i \leq t_B$, and so $a_i \leq t_i \leq t_i + d_i \leq t_B \leq b_B \leq a_i + P$. Because A_i are valid and stable on time interval $[t_i, t_B[$ (by hypothesis), all inputs of B are valid at time t_B. 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 $[t_i, b_B[$ (by hypothesis). On time interval $[b_B, a_B + P[$, 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 Fig. 7 (additions taking one time unit and multiplications taking two). We should notice that the multiplication (v1) is longer than the period (P). If $d(v) > P$, we have that $s_n(v) + d(v) > s_{n+1}(v)$, which means that we must start the next calculation in vertex v before the current one finishes. There are two ways to accomplish this: pipeline v (Fig. 7b) or put multiple instances of v (Fig. 7c). To be able to pipeline v, we must ask the designer (or a synthesis tool) to split the calculations in v so that each part has a delay $\leq P$.

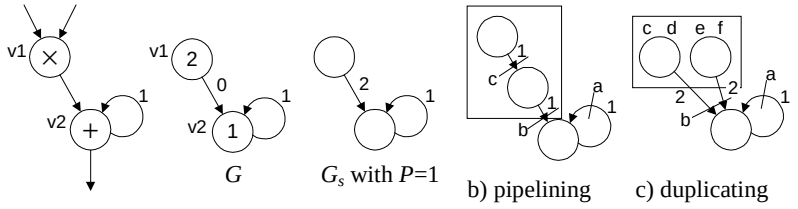


Fig. 7. Scalar product example

To put multiple instances of v, we can modify `BreakPath` so that when $d(v) > P$ it places $\lceil \frac{d(v)}{P} \rceil$ functional elements like v, each one with registered inputs and multiplexing the outputs. At each period, the input registers from the next unit are enabled and the multiplexor chooses the right unit for output. We must adjust $d(v)$ 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.

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

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

5.2 Fractional clock duration and k-periodic scheduling

Suppose we want to find an optimal circuit for the circuit graph G in Fig. 8; $P = 5/2$ is fractional and the figure shows the register placement obtained from `BreakPath`. A consistent schedule is to be found in the left part of Table 5. Notice that c_1 and c_2 are but one register c: both have same input and schedule. This first solution may be acceptable but it requires a clock resolution smaller than the time unit used. If all the $d(v)$ are integers then there exists an optimal valid schedule with $s_n(v)$ being all integers. In fact, if all the $d(v)$ are integers, a theorem [Bennour and Aboulhamid

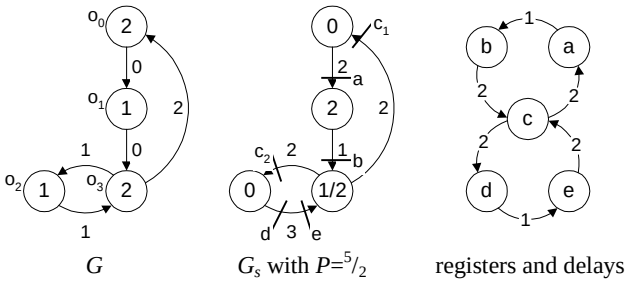


Fig. 8. k-periodic example

1995; Van Dongen et al. 1992] says that if we have a valid periodic schedule s then the schedule s^* defined as $s^*_n(v) = \lfloor s_n(v) \rfloor$ is a valid k-periodic schedule with the same throughput. Applying this result to our first schedule gives the 2-periodic schedule with period $2P = 5$ shown in Table 5: during each period, two output values are calculated, so giving the same throughput as the first schedule.

	periodic ($P = 5/2$)	*k-periodic* ($k = 2$)		
name	***enable time***	***schedule*$_1$**	***schedule*$_2$**	***enable times***
a	$[1/2, 2[$	2	4	$[3, 2[$
b	$1/2$	3	0	$\{0, 3\}$
c	$[2, 1/2[$	0	2	$\{0, 2\}$
d	$[1/2, 2[$	2	4	$[3, 2[$
e	$1/2$	3	0	$\{0, 3\}$

Table 5. A schedule for the solution of Fig. 8, and a corresponding k-periodic schedule.

6. EXPERIMENTATION AND IMPLEMENTATION

For our experimentation, we have used 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 [Fraser and Hanson 1995]. The first benchmark is the one presented in [Leiserson and Saxe 1991; Lockyear and Ebeling 1994]. In order to compare our results we also implemented the original retiming method of Leiserson and Saxe [1991]. From Table 6 we see that 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 [Benchmarks 1996]. It is interesting to note that the elliptic filter specification in the suite cannot be accelerated, but by re-writing the specification, we obtain an acceleration of 150% using retiming and an additional 9% using our method. The original description used only additions, but after flattening the expressions, we see that the same variable is added multiple times, which could be replaced by a multiplication by a constant. The multiplication

can then be divided in shifts and additions, shifts taking no time as it is only a different interconnection. Then, tree balancing of the expressions was used, to reduce the length of the critical path.

To compare the running time of our algorithm to that of retiming, we ran them on benchmarks from ISCAS89 [Benchmarks 1996]. These circuits are at gate level and have up to 16K gates. We see, in Table 7, that we are around 10 times faster than SIS, except for the larger circuit on which we are 185 times faster. The algorithm used to find maximum throughput has a running time lower than the binary search approach presented in Section 3.1, although the upper bound is not clear.

	Period		***Registers****		***Phases***	***Acceleration***
	Retiming	***L.A.***	***Retiming***	***L.A.***		
correlator	13	10	5	4.5	2	30%
scalar product	2	1	2	4	2	100%
k-periodic	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%

Table 6. Benchmarks.
*Register count is the number of edge-triggered plus $^1/_2$ the number of level-sensitive storage.

	Period			***Computation time***	
	Original	***Retiming***	***L.A.***	***Retiming***	***L.A.***
s344	18.8	14.2	14.2	0.8	0.12
s641	30.4	-	-	5.8	0.29
s713	41.6	-	-	7.9	0.38
s1238	28.4	-	-	49.8	1.81
s1423	82.6	73.4	73.4	34.4	2.02
s1488	34.8	32.4	31.9	22.9	3.09
s1494	34.8	32.8	32.8	32.5	4.16
s5378	19.2	14.6	14.6	89.8	15.37
s9234	44.0	-	-	81.7	13.98
s13207	58.4	37.6	36.5	735.8	85.67
s15850	80.4	44.8	44.8	772.4	103.46
s35932	25.0	-	-	3.8 days	30 minutes

Table 7. Computation time (in seconds) of retiming by SIS (retime -ni) vs. scheduling by L.A., on a UltraSPARC-10 with 128MB. A dash says that no optimizations have been made.

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 proved computing complexity is similar to previously published methods but on the benchmarks we are much faster and 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 are placed by a greedy

algorithm. Future work includes the design of an optimal algorithm to maximize chaining and minimize the number of clock phases and of registers. 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 allow 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

Benchmarks. 1996. North Carolina State University, Dep. of Comp. Science, Collab. Benchmark Lab., `http://www.cbl.ncsu.edu/benchmarks/Benchmarks-upto-1996.html`.

Bennour, I. E. and Aboulhamid, E. M. 1995. Les problèmes d'ordonnancement cycliques dans la synthèse de systèmes numériques. Technical Report 996 (Oct.), DIRO, Université de Montréal. `http://www.iro.umontreal.ca/~aboulham/pipeline.pdf`.

Deokar, R. B. and Sapatnekar, S. 1995. A fresh look at retiming via clock skew optimization. In *DAC'95* (1995), pp. 304–309.

Van Dongen, V. H., Gao, G. R., and Ning, Q. 1992. A polynomial time method for optimal software pipelining. In *CONPAR'92, Lecture Notes in Computer Sciences, Vol 634* (1992), pp. 613–624.

Fraser, C. W. and Hanson, D. R. 1995. *A Retargetable C Compiler: Design and Implementation*. Benjamin Cummings.

Hanen, C. 1994. Study of a NP-hard cyclic scheduling problem: the recurrent job-shop. *European Journal of Operation Research 72*, 1, 82–101.

Hartmann, M. and Orlin, J. 1993. Finding minimum cost to time ratio cycles with small integral transit times. *Networks; an international journal 23*, 1, 82–101.

Hwang, C.-T., Hsu, Y.-C., and Lin, Y.-L. 1991. Scheduling for functional pipelining and loop winding. In *DAC'91* (1991), pp. 764–769.

Ishii, A. T., Leiserson, C. E., and Papaefthymiou, M. C. 1997. Optimizing two-phase, level-clocked circuitry. *Journal of the ACM 44*, 1 (Jan.), 148–199.

Lawler, E. 1976. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing.

Legl, C., Vanbekbergen, P., and Wang, A. 1997. Retiming of edge-triggered circuits with multiple clocks and load enables. In *IWLS'97* (1997).

Leiserson, C. E. and Saxe, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 1, 3–35.

Lockyear, B. and Ebeling, C. 1994. Optimal retiming of level-clocked circuits using symmetric clock schedules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 9 (Sept.), 1097–1109.

Maheshwari, N. and Sapatnekar, S. 1997. An improved algorithm for minimum-area retiming. In *DAC'97* (1997), pp. 2–6.

Maheshwari, N. and Sapatnekar, S. 1998. Efficient retiming of large circuits. *IEEE Transactions on VLSI Systems 6*, 1 (March), 74–83.

De Micheli, G. 1994. *Synthesis and optimization of digital circuits*. McGraw-Hill.

Sakallah, K. A., Mudge, T. N., and Oluktun, O. A. 1990. Analysis and design of latch-controlled synchronous digital circuits. In *DAC'90* (1990), pp. 111–117.