# Combining Retiming and Recycling to Optimize the Performance of Synchronous Circuits

Luca P. Carloni Alberto L. Sangiovanni-Vincentelli
EECS Department, University of California at Berkeley, Berkeley, CA 94720-1772

## Abstract

*Recycling was recently proposed as a system-level design technique to facilitate the building of complex System-on-Chips (SOC) by assembling pre-designed components. Recycling allows us to model the communication patterns among the components, analyze the impact of interconnect latency on the overall data processing throughput, and manage computation/communication tradeoffs to optimize the performance of the system. In this paper, we present recycling as a circuit-level design technique for optimizing the performance of sequential circuits beyond what can be achieved by retiming. We also provide a theoretical framework to guide the simultaneous application of the two techniques. Our model identifies the conditions under which an optimally-retimed synchronous circuit can be further sped-up and determines the amount of the resulting performance gain.*

## 1 Introduction

*Recycling*, originally proposed for system-level design [6], is a design transformation that can be applied to a class of systems called latency-insensitive systems [5]. It consists of inserting proper sequential elements (*relay stations*) on the communication channels interconnecting the system components while gauging latency variations and impact on overall system performance. We show how recycling can be used to increase the performance of a synchronous digital circuit at the register-transfer level (RTL). In particular, we combine recycling to *retiming*, a classical logic optimization technique to minimize the clock period of a synchronous circuit [12]. Recycling goes beyond what retiming offers because:

- it can be applied to a network of sequential circuits, i.e. a circuit whose computational elements are sequential circuits, while retiming can deal only with a single sequential circuit whose computational elements are combinational circuits;
- it can be applied to retimed circuits to obtain further speed-up; this is achieved by extra-pipelining the circuit through relay-station insertion, thereby circumventing the main limiting factor in retiming, namely the presence of feedback loops.

The first item is extensively discussed in [5, 6], albeit from a different perspective. Here, we cast the arguments in the context of RTL design. This paper's main contribution addresses the second item. We discuss how to transform a synchronous circuit into a latency-insensitive design and how to apply recycling to break the critical paths and reduce further the clock period. We give precise criteria on when this operation pays off from a performance viewpoint.

After providing some background material on retiming (Section 2) and recycling (Section 3), we use a simple example to illustrate the idea of jointly applying recycling and retiming (Section 4). This allows us to introduce application criteria for recycling that are only based on the circuit topology and the delay characteristics of its components. Recycling does not come for free, but implies an overhead in terms of both area and latency. In fact, there are cases when this transformation may not be convenient even in terms of timing optimization. Fortunately, these cases can be detected in a relatively easy way. We provide a formula to estimate the impact of recycling on the circuit speed (Section 5). Finally, we put our contribution in perspective with previous papers that attempt to *"go beyond retiming"* (Section 6). In particular, we compare recycling to *c-slow retiming*, or *slowdown*, a technique that was originally presented together with retiming [11] and, then recently applied in the context of FPGA design [17, 19]. Recycling and slowdown are strictly related, although they target different applications. While slowdown is best applied to circuits operating on multiple input data streams, we argue that, for the case of synchronous circuits operating on a single input data stream, recycling subsumes $c$-slow retiming, in the sense that the timing optimization obtained by the latter can always be obtained by the former but not vice versa. The relationship between retiming and software pipelining has been studied in several works. In general it is retiming that is applied to further enhance software pipelining [4, 7], but a recent work proposed software pipelining techniques as a better alternative to retiming for sequential circuit optimization [2, 3]. We discuss this idea with respect to our contribution, by pointing out the theoretical similarities and the practical differences.

## 2 Leiserson and Saxe's Retiming

Retiming is a *classic* logic optimization technique for synchronous circuits. Retiming was originally introduced in [9, 10, 11], where the emphasis is put on the application to systolic systems. A subsequent paper [12] fully revisits the concept of retiming and shows how generic synchronous circuits can benefit from it under three main optimality criteria: (1) minimize the circuit clock period by adding/removing storage elements (this is the main focus of the present paper); (2) minimize the circuit area by reducing the

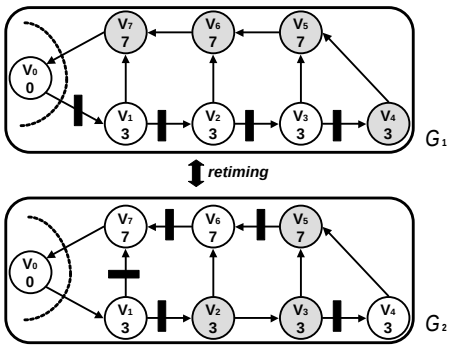


**Figure 1. Retiming the correlator circuit. Top graph has $\pi(\mathcal{G}_1) = 24$, bottom one has $\pi(\mathcal{G}_2) = 13$ (shading highlights critical paths).**

number of storage elements; and, finally, (3) minimize circuit area under a maximum clock-period constraint. In the last two decades, retiming has been adopted as a key optimization technique within every major logic synthesis tool both in academia and industry. In [15, 16], Shenoy discusses the issues that arise in practical implementations of retiming and the research efforts to extend the domain of circuits for which it can be applied. Recently, retiming has been successfully applied to FPGA design [8, 17, 19].

To review the main concept of retiming, let's consider Figure 1, which reports an example from [12]. The two graphs represent two synchronous circuits that are functionally equivalent and can be obtained one from the other via retiming. *Functional equivalence* means that the circuits have the same behavior from the host system viewpoint, i.e. when solicited by the same input trace they present exactly the same output trace. We define a *circuit graph* as a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{A}, d, w)$, where $\mathcal{V}$ is the set of vertices (the combinational circuits), $\mathcal{A}$ is the set of arcs (the wires between combinational circuits), $d$ is a vertex labelling function (the largest combinational delay of the corresponding circuit) and $w$ is an arc weight function (the number of storage elements on the arc). Let $\mathcal{C}(\mathcal{G})$ be the set of cycles of $\mathcal{G}$. For $\mathcal{G}$ to have well-defined physical meaning, the following constraints must be satisfied: $\forall v \in \mathcal{V}\big(d(v) \geq 0\big)$, $\forall a \in \mathcal{A}\big(w(a) \geq 0\big)$, and $\forall c \in \mathcal{C}(G), \exists a \in c \; s.t. \; w(a) > 0$.

The circuit represented by the graphs of Figure 1 is a digital correlator that takes a stream of bits $x_i$ and compares it with a fixed-length pattern $a_1, \ldots, a_J$ to produce the output $y_i = \sum_{j=0}^{J} f(x_{i-j}, a_j)$, where $f(x,y)$ is the comparison function returning one if $x = y$ and zero otherwise. Circuit graph $\mathcal{G}_1$ is a direct implementation of this specification for $J = 3$. Its components are instances of two kinds of combinational elements: 4 comparators ($v_1, v_2, v_3, v_4$) and 3 adders ($v_5, v_6, v_7$), while vertex $v_0$ represents the host system. This implementation has 4 storage elements (e.g., edge-triggered flip-flops controlled by a common clock) represented by the dark rectangles. For the delay assignment of the combinational elements, we initially take the same figures as in [12]: 7 time units for the adders and 4 time units for the comparators.

For any path $p \in \mathcal{G}$ of length $|p| = k$, that is an alternating sequence of vertices and arcs $v_1, a_1, \ldots, a_{k-1}, v_k$, we define the *path delay* $d(p)$ and the *path weight* $w(p)$ respectively as $d(p) = \sum_{i=1}^{k} d(v_i)$ and $w(p) = \sum_{i=1}^{k-1} w(a_i)$. For instance, graph $\mathcal{G}_1$ in Figure 1 has a path $p$ from $v_1$ to $v_0$ passing through $v_4$ with $d(p) = 33$ and $w(p) = 4$.

A *combinational path* $p$ is a path that does not contain any storage element, i.e. $w(p) = 0$. The longest combinational path (*critical path*) in a synchronous circuit is a lower-bound on the minimum value of the clock period $\pi(\mathcal{G})$ at which the circuit can operate. This lower-bound is strict if we assume that the clock overhead can be neglected, otherwise we should include the value of the expression $\pi_{overhead} = \pi_{skew} + \pi_{jitter} + \pi_{latch}$ containing the estimations of clock skew, clock jitter, and latch set-up time. The critical path $p_{crit}$ of $\mathcal{G}_1$ is the path from $v_4$ to $v_7$ with $d(p_{crit}) = 24$.

A *retiming* of a circuit graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}, d, w)$ is an integer-valued vertex-labelling $r : \mathcal{V} \rightarrow \mathbf{Z}$ specifying a graph transformation that returns a new graph $\mathcal{G}_r = (\mathcal{V}, \mathcal{A}, d, w_r)$ such that:

$$\forall a = (v_i, v_j) \in \mathcal{A}, \Big(w_r(a) = w(a) + w(v_j) - w(v_i)\Big)$$

We refer to [11] for the proof of correctness of retiming. Graph $\mathcal{G}_2$ in Figure 1 represents the result of applying retiming for optimal clock period on $\mathcal{G}_1$. By comparing the two graphs, one can verify that $\mathcal{G}_2$ is obtained by repositioning two flip-flops and adding one new flip-flop according to the vertex-labelling $r$ such that $r(v_0) = 0, r(v_1) = -1, r(v_2) = -1, r(v_3) = -2, r(v_4) = -2, r(v_5) = -2, r(v_6) = -1, r(v_7) = 0$. Hence, retiming reduces the clock period to $\pi(\mathcal{G}_2) = 13$ units. In fact, retiming enables interesting trade-offs between area (the circuits differ by one flip-flop) and performance ($\mathcal{G}_2$ gives a 45% speed-up). Circuit retiming for clock period minimization can be casted as a linear programming problem, which can be solved efficiently in $\bigcirc(V^3)$ steps with $\bigcirc(V \cdot A)$-time Bellman-Ford shortest-path algorithm. An asymptotically faster algorithm running in $\bigcirc(V \cdot A)$ time is given in [12].

It is easy to see that graph cycles (i.e. circuits feedback loops) are a limiting factor of retiming. As proven in [12], all the possible retimed versions of a circuit must satisfy the following *invariant rule*: the number of storage elements that lie on any given cycle $c$ in the graph $\mathcal{G}$ remains constant through retiming, or, formally:

$$\forall c \in \mathcal{G}\Big(w_r(c) = w(c)\Big) \tag{1}$$

Further, as proven by M. Papaefthymiou [14], there exists a theoretical lower-bound on the minimum clock period achievable via retiming and this is a function of the *maximum delay-to-register cycle ratio*. This is defined as the maximum value (over all cycles) of the ratio between the sum of combinational delays on the cycle and the sum of storage elements on the cycle, or, formally:

$$\pi_{lb}(\mathcal{G}) = \left\lceil \max_{c \in \mathcal{C}(\mathcal{G})} \pi(c) \right\rceil = \left\lceil \max_{c \in \mathcal{C}(\mathcal{G})} \left( \frac{\sum_{v \in c} d(v)}{\sum_{a \in c} w(a)} \right) \right\rceil \tag{2}$$

We call *critical cycle* any cycle $c \in \mathcal{C}(\mathcal{G})$ such that $\pi(c) = \pi_{lb}(\mathcal{G})$. The lower-bound is theoretical because it may not be reached by a retiming transformation. For instance, the minimum clock period achievable via retiming for the circuit of Figure 1 is $\pi(\mathcal{G}) = 13$, while the theoretical lower-bound is $\pi_{lb}(\mathcal{G}) = 10$, as it can be verified by computing $\pi(c)$ for the 4 cycles of the circuit. Finally, as proven in [14], it exists also a corresponding upper-bound that is given by $\pi_{ub}(\mathcal{G}) = \pi_{lb}(\mathcal{G}) + d_{max} - 1$, where $d_{max} = \max_{v \in \mathcal{G}} d(v)$.

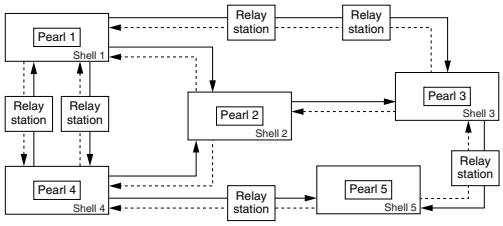


**Figure 2. Recycling a latency-insensitive system.**

## 3 Latency-Insensitive Protocols and Recycling

*Recycling* was recently proposed as the basis of a correct-by-construction methodology to quickly assemble Intellectual Properties (IP) cores for building complex System-on-Chips (SOC) while reducing the number of iterations in the design process [6]. Its foundations lie on the *theory of latency-insensitive design* [5]. Latency insensitive systems are synchronous distributed systems realized by composing functional modules that exchange data on point-to-point communication channels according to a specific protocol. This latency-insensitive protocol guarantees that each module works independently of the channel latencies, thereby making the overall system functionality robust with respect to those insidious arbitrary variations in interconnect delay that are proper of SOCs fabricated with deep sub-micron technologies [1, 13, 18]. The protocol works on the assumption that the modules are *stallable*, a weak condition to ask them to obey. Figure 2 illustrates a latency-insensitive system. Following the jargon of [5], we have that the system modules (the *pearls*) are encapsulated by interface modules (the *shells*) that *speak* the latency-insensitive protocol. A latency-insensitive system can be *recycled* by adding *relay stations* on its channels. A relay station is a sequential circuit (a little bit more complex than a normal flip-flop) that effectively pipelines the interconnect wire on which it is placed.

A *recycling* of a circuit graph $\mathcal{G} = (\mathcal{V}, \mathcal{A}, d, w)$ is an integer-valued arc-labelling $s : \mathcal{A} \rightarrow \mathbf{Z}^*$ that returns a recycled graph $\mathcal{G}_r = (\mathcal{V}, \mathcal{A}, d, w, s)$ where $s(a)$ denotes the number of relay stations on arc $a$. Observe that $\mathcal{G}$ now represents a latency-insensitive system, i.e. each vertex in $\mathcal{G}$ is associated to a pearl/shell pair. For the purposes of SOC design, the recycling transformation consists in pipelining long and slow communication channels in shorter and faster stages to make sure that they can be driven at the same clock frequency of the rest of the system (dictated by the slowest pearl). More generally, recycling is a technique to re-organize the system structure while efficiently exploring the design space and the trade-offs between computation delay and communication latency. In the next section we study how to apply these ideas at the circuit level, but, before, we need to clarify two important points:

- As long as they are stallable, the pearl modules can be arbitrary sequential circuits and, therefore, contain various storage elements, and implement complex finite state machines (FSM). This is an important difference with respect to retiming, which operates on sequential elements across *combinational circuits*. In other words, retiming can be applied only to a graph $\mathcal{G}$ whose vertices are combinational circuits.
- The theory of latency-insensitive protocols guarantees that after performing shell encapsulation on a system $S$, the resulting latency-insensitive system $S'$ is functionally equivalent to the original one modulo *stuttering*. This means that if $S'$ is solicited with the same input traces as $S$, it produces an output trace that presents exactly the same ordered sequence of data as $S$, but where two consecutive valid data may be interleaved by one or more stalling events (a.k.a. $\tau$-symbol events, or *bubbles*). Naturally, the stalling events can be easily filtered out to obtain *exactly* the same output trace. The nature of the stalling events goes back to the idea of shells and relay stations. The simplest way to insert extra sequential elements into a synchronous circuit without jeopardizing its functional behavior, is to make sure that they are initialized with values that do not disrupt the state of the down-link sequential processes. Hence, each relay station is initialized with a $\tau$ symbol (this can be done via special encoding or by adding an extra wire to a given bus). Then, the interface logic of a shell simply operates according to an *AND firing rule* meaning that if *all* the input channels present valid data, then it lets the pearl to receive them, progress its computation, and produce valid output data, but if *even a single* channel presents a $\tau$, then the shell stalls the pearl and puts new $\tau$ symbols on the output channels instead of valid data. For more details on how the latency-insensitive protocol operates and the role of *back-pressure* we refer to the original work [5, 6].

## 4 Combining Recycling And Retiming

From the previous section we understand that recycling and latency-insensitive protocols allow us to insert sequential elements (relay stations) on any wire of a latency-insensitive system. Now, our idea is based on the following considerations: (1) any stand-alone combinational circuit can be made sequential by inserting one storage element on each of its outputs, and (2) the resulting sequential circuit is a stallable circuit that can be made patient by synthesizing a shell around it. Hence, we can transform a retimed circuit into a latency-insensitive system so that we can pipeline its critical paths via relay station insertion, and, ultimately, reduce the overall clock period. In other words, we use the theory of recycling and latency-insensitive protocols to *break* the retiming invariant rule expressed by Equation (1).

Let's consider the circuit graph $\mathcal{G}_3$ pictured at the top of Figure 3, a recycled version of the correlator that can be derived with the following procedure: (1) apply a new retimed transformation to the optimally-retimed graph circuit graph $\mathcal{G}_2$ of Figure 3 by using a vertex-labeling $r'$ such that $r'(v_5) = 1$ and $r'(v_6) = 1$ while $r'(v_j) = 0$ for $j \neq 5, 6$ (notice that the critical path of the resulting circuit is the path connecting $v_6$ to $v_1$ for a delay of 17 time units); (2) partition the circuit in sub-circuits whose outputs are delimited by storage elements and create a shell around each sub-circuit; (3) insert a relay station between vertices $v_6$ and $v_7$ to break the critical path. Knowing that relay stations are sequential elements, it is easy to verify that the critical path of the resulting graph $\mathcal{G}_3$ is the path connecting $v_7$ to $v_1$ for a length of 10 time units. Hence, in first approximation, $\mathcal{G}_3$ can nominally run with a clock period $\pi(\mathcal{G}_3) = 10$, a 23% speed-up with respect to $\mathcal{G}_2$. We can push this

idea to its limits by considering circuit graph $\mathcal{G}_4$ at the bottom of Figure 3. Here, by inserting a new relay station between $v_7$ and $v_8$, we break further the critical path to reach the minimum possible length. This coincides with the delay of the slowest circuit component, in this case the adder, which is 7 time units. Hence, $\mathcal{G}_4$ can run with a nominal clock period as small as $\pi(\mathcal{G}_4) = 7$, a 46% speed-up with respect to $\mathcal{G}_2$.

Besides having such considerable nominal speed-ups with respect to the optimally-retimed circuits, it would appear that the ability of inserting extra storage elements, thus overcoming the retiming invariant rule, allows us to beat even the theoretical lower-bound expressed by the maximum delay-to-register cycle ratio of Equation (2). But, before making such claims, more considerations need to be done. First, it is not realistic to assume that the insertion of shell logic does not cause some timing overhead. To take this into account, we should include an additional term $\pi_{shell}$ to the other components of the clock overhead $\pi_{overhead}$. More importantly, even if we could neglect the impact of $\pi_{shell}$ with respect to the critical path delay, it is the presence of the $\tau$ symbols that affects negatively the amount of speed-up achievable with recycling. In fact, after initializing each relay station with a $\tau$ symbol, the latency-insensitive protocol's *AND firing rule* implies that the number of $\tau$ symbols in a cycle of the circuit remains constant during its operations. Hence, $\tau$ symbols are periodically detected at the circuit outputs and, since we cannot consider them as valid results, they must be discounted from the assessment of the circuit performance. Let's define the circuit throughput $\vartheta(\mathcal{G})$ as the ratio of valid data over the sum of valid data plus $\tau$ symbols (as observed at the circuit's outputs). Since the recycled circuit is a latency-insensitive system, this ratio can be statically computed as the minimum value (over all cycles) of the ratio between the sum of storage elements (that are not relay stations) and the sum of all storage elements (including also the relay stations) or, formally:

$$\vartheta(\mathcal{G}) = \min_{c \in \mathcal{C}(\mathcal{G})} \left( \frac{\sum_{a \in c} w(a)}{\sum_{a \in c} w(a) + s(a)} \right) \tag{3}$$

Notice that $\vartheta(\mathcal{G})$ is a rational number between zero and one. Then, We define the *effective clock period* $\pi_{eff}(\mathcal{G})$ of a recycled circuit as the ratio of the nominal clock period over the circuit throughput:

$$\pi_{eff}(\mathcal{G}) = \frac{\pi(\mathcal{G})}{\vartheta(\mathcal{G})} \tag{4}$$

Hence, if we consider the effective clock period instead of the nominal one, we see that, even if we neglect $\pi_{shell}$, recycling does not really pay off for the examples of Figure 3. In particular, circuit $\pi(\mathcal{G}_3)$ has $\vartheta(\mathcal{G}_3) = 2/3$, and, therefore, $\pi_{eff}(\mathcal{G}_3) = 10 \cdot (3/2) = 15$, while circuit $\pi(\mathcal{G}_4)$ has $\vartheta(\mathcal{G}_4) = 1/2$, and, therefore, $\pi_{eff}(\mathcal{G}_4) = 7 \cdot (2/1) = 14$. The effective performance of both circuits is beaten by the optimally-retimed circuit $\pi(\mathcal{G}_2)$ having clock period $\pi(\mathcal{G}_2) = 13$.

In summary, we saw that we succeeded in using recycling to overcome the *retiming invariant rule* just to find ourselves in the position of dealing with something like a "*recycling invariant rule*" that has canceled all the gain. Does this mean that recycling can never give us a circuit implementation whose performance is better than the optimally-retimed version? It is easy to prove that this is not the case. A counter-example is given by the same correlator circuit with a different delay assignment. For instance, if assume that all vertices have the same delay $\delta$ but for $d(v_0) = 0$, then we can prove that the recycled circuit graph $\mathcal{G}_4$ has an effective clock period $\pi_{eff}(\mathcal{G}_4) = (2/1) \cdot \delta = 2 \cdot \delta$, which is strictly smaller than the clock period of the optimally-retimed graph for every possible value of $\delta$. In fact, since retiming cannot change the number of flip-flops on a cycle, each cycle $c$ of any retimed correlator will present a sequence $\sigma_m(c)$ of $m$ consecutive vertices whose connecting arcs have zero weight (i.e. the arcs do not present a storage element) where $m = \lceil \frac{|c|}{w(c)} \rceil$. In particular, while cycles $c_0 = v_0, v_1, v_2$ and $c_4 = v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$ present sequences $\sigma_m(c_i)$ of at most length 2, both cycles $c_1 = v_0, v_1, v_2, v_6, v_7$ and $c_2 = v_0, v_1, v_2, v_3, v_5, v_6, v_7$ present a sequence $\sigma_m(c_i)$ of length 3. Hence, under this delay assignment, any optimally-retimed circuit $\mathcal{G}$ will have a critical path of 3 vertices for a minimum clock period $\pi(\mathcal{G}) = 3 \cdot \delta$, which is 33% worse than the effective clock period $\pi_{eff}(\mathcal{G}_4)$ of the corresponding optimally-recycled circuit. Observe that the 33% gain that we obtain with recycling remains constant if we extend the length of the correlator by considering any implementation with $J > 3$. In fact, notice that the extension would create new cycles presenting critical path of 3 vertices, while the effective clock period of the recycled circuit depends only on $\delta$.

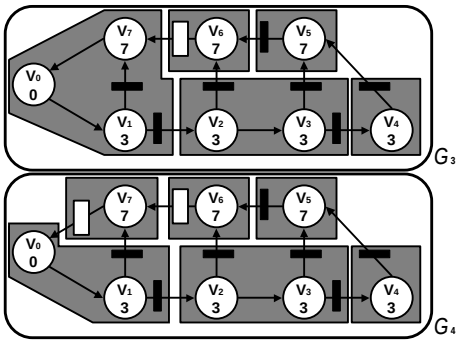


**Figure 3. Two recycled versions of the correlator (shading shows the shell wrapping; light rectangles are relay stations initialized to $\tau$).**

In conclusion, there are cases when recycling does not produce any gain with respect to an optimally-retimed circuit and cases where a relevant gain is guaranteed. This depends on the circuit topology and its delay assignment. Next we give a general model to analyze when combining recycling and retiming pays off.

## 5 Recycling Benefit Analysis

From the analysis of the combination of retiming and recycling for the correlator circuit it should be clear that the key point is to understand the structure of the cycles of a circuit. After all, Equations (1), (2), and (3) are all related to circuit cycles. Hence, in the attempt of providing an analytical model for the combined application of recycling and retiming, let's consider the two circuits of

Figure 4. Both circuits present a single cycle on which $N$ vertices sit. The top circuit, $\mathcal{G}_5$, has $N-1$ registers while in the bottom circuit, $\mathcal{G}_6$, there is only 1 register. These represent two extreme cases for this circuit structure, which, generally, can have $K$ registers with $K \in [1, N-1]$. Notice that the case $K = 0$ is ruled out because we assume the absence of combinational loops. Also case $K \geq N$ is not interesting because the application of either retiming or recycling would not produce any benefit from the timing optimization viewpoint (the circuit would run with a clock period equal to the maximum combinational delay among all its components anyway).

Now, for $K \in [1, N-1]$ we know that the theoretical lower bound on the minimum clock period achievable via retiming is $\pi_{lb}(\mathcal{G}) = \lceil \frac{\sum_{i=1}^{N} d(v_i)}{K} \rceil$. In Figure 4, this gives $\pi_{lb}(\mathcal{G}_5) = \lceil \frac{\sum_{i=1}^{N} d(v_i)}{N-1} \rceil$ and $\pi_{lb}(\mathcal{G}_6) = \sum_{i=1}^{N} d(v_i)$. Also, we know that the optimally-retimed graph presents a sequence $\sigma_m^\star$ of $m$ consecutive vertices without registers in between, where $m = \lceil \frac{N}{K} \rceil$ and the '$\star$' symbol denotes that this sequence is the one with the smallest *delay sum* among the $N$ possible ones. We write this delay sum as $d(\sigma_m^\star)$ (it coincides with the circuit clock period if no single vertex has delay greater than this). Hence, $\pi_{ret}(\mathcal{G}_5) = \max\{d_{max}, (d(v_x) + d(v_y))\}$, where $v_x, v_y$ are the two consecutive vertices with the smallest compound delay, while $\pi_{ret}(\mathcal{G}_6) = \sum_{i=1}^{N} d(v_i)$. Now, let's denote with $H$-recycling the insertion of $H$ relay stations on the circuit. The value of $H$ depends on both $N$ and $K$, as well as the vertex delay assignment, but, in any case, it is useless to have $H > N-K$. Now, it is easy to see that the nominal clock period of a $H$-recycled circuit is $\pi_H(\mathcal{G}) = \max\{d_{max}, d(\sigma_{m'}^\star)\}$ with $m' = \lceil \frac{N}{K+H} \rceil$. For instance, we have that $\pi_1(\mathcal{G}_5) = d_{max}$. In general, the nominal clock period $\pi_H(\mathcal{G})$ is smaller than the corresponding $\pi_{ret}(\mathcal{G})$, but, to find the effective clock period we must divided it by the recycled circuit throughput $\vartheta = \frac{K}{K+H}$, Putting all together, we conclude that recycling returns a gain over optimal retiming if and only if

$$\max\{d_{max}, d(\sigma_{m'}^\star)\} \cdot \frac{K+H}{K} \; < \; d(\sigma_m^\star) \qquad (5)$$

The gain percentage is given by the following:

$$gain(\%) \; = \; \left(1 - \frac{\max\{d_{max}, d(\sigma_{m'}^\star)\} \cdot \frac{K+H}{K}}{d(\sigma_m^\star)}\right) \cdot 100 \qquad (6)$$

If we want to take into account also the clock overhead due to the shell logic then we simply need to add the term $\pi_{shell} \cdot \frac{K+H}{K}$ to the left-hand side of the inequality.

For the special case of delay-homogeneous circuits (where $\forall i, d(v_i) = \delta$) the previous condition can be simplified as follows:

$$\left\lceil \frac{N}{K+H} \right\rceil \cdot \delta \cdot \frac{K+H}{K} < \left\lceil \frac{N}{K} \right\rceil \cdot \delta \Leftrightarrow \left\lceil \frac{N}{K+H} \right\rceil \cdot \frac{K+H}{K} < \left\lceil \frac{N}{K} \right\rceil \quad (7)$$

From this, it is easy to see that recycling never pays off for circuit $\mathcal{G}_6$ while, as long as $N > 2$, it always does for circuit $\mathcal{G}_5$. In the latter case the gain percentage is given by $\frac{N-2}{2 \cdot N-2} \cdot 100$, which varies from 25% for $(N = 3)$ up to a maximum of 50%. Also, notice that for circuit $\mathcal{G}_5$ recycling *technically* beats the theoretical lower bound of Equation (2) as it allows to match exactly the ratio $\frac{N}{N-1} \cdot \delta$, instead of its integer ceiling. Finally, observe that the model presented in this section can be used to efficiently analyze more complex circuits containing many intersecting cycles because Equation (5) still applies for each potential critical cycle.

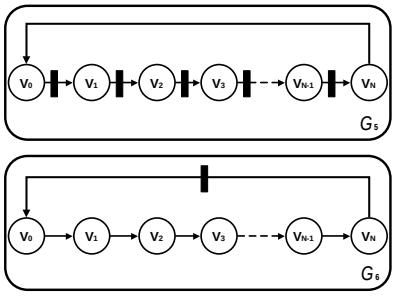


**Figure 4. Case studies: $N$-1 FFs (top), 1 FF (bottom).**

## 6 Recycling vs. Alternative Approaches

In this section we briefly discuss the relationship between recycling and two other techniques that have been proposed to overcome the retiming invariant rule. Slowdown, or *c-slow retiming*, was proposed together with retiming in [12], where the main goal is to establish format techniques to transform a synchronous circuit into a functionally-equivalent systolic circuit, i.e. a circuit that presents at least one register on every arc of its circuit graph. Slowdown is performed in two steps: (1) replace each register in the original circuit by a sequence of $c$ registers, thus producing the $c$-slow equivalent circuit; (2) retime the $c$-slow circuit to minimize its clock period. While slowdown is obviously defined for $c$ being an integer greater than one, it should be noticed that for any given circuit $\mathcal{G}$, there is an integer $\gamma(\mathcal{G})$ such that $c$-slow retiming produces a clock period reduction if and only if $c \geq \gamma(\mathcal{G})$. Furthermore, slowdown does not come for free, but, besides the extra area due to additional registers, it implies a performance overhead, somewhat similar to recycling. In particular, as suggested by its very name, a $c$-slow circuit, while running with a potentially higher frequency clocks, processes a single input data at the throughput of $1/c$. In fact, at any given clock iteration, only $1/c$ of the registers in the $c$-slow circuit contains valid data. Hence, if $\pi^c$ is the clock period achievable via slowdown, the effective clock period is $\pi^c_{eff} = \pi^c/c$. It should not be a surprise, then, that slowdown was proposed with the idea of contemporaneously processing $c$ input data streams by properly multiplexing and demultiplexing the I/O ports of the $c$-slow circuit. By doing so, the processing throughput can be raised back (up to one in the optimal case, where each computation thread operates with period $\pi^c_{eff}$).

For the purpose of this paper, we want to clarify how, besides the obvious similarities, recycling and slowdown differ and, particularly, how the former subsumes the latter when one considers only the processing of single input data stream. In this case, recycling can always mimic slowdown, because the same results obtained with a $c$-slow retiming can be obtained via recycling by inserting relay stations (up to $c-1$ per cycle) instead of duplicating registers. In both cases, the resulting circuit throughput is given by $1/c$. On the other side, it is not difficult to find an example of a case where a slowdown can not achieve the same performance that recycling offers due to the coarser granularity of the slowdown

transformation. For instance, consider again circuit $\mathcal{G}_5$ of Figure 4 representing a feedback loop with $N$ vertices $v_i$, each with a delay $d(v_i) = \delta$, and $N-1$ registers. The insertion of one single relay station produces a recycled circuit with a nominal clock period equal to $\delta$ and a circuit throughput equal to $\frac{N-1}{N}$ for an effective clock period $\pi_{eff} = \frac{N}{N-1} \cdot \delta$. Instead, the simplest slowdown transformation produces a 2-slow circuit that has the same clock period $\delta$ as the recycled circuit, but a smaller throughput equal to $\frac{1}{2}$ (which effectively cancels the benefits of the transformation because $2 \cdot \delta$ is the clock period of the original circuit). Hence, as long as $N > 2$, the performance of the recycled circuit is better than the performance of the corresponding 2-slow circuit by a percentage equal to $\frac{N-2}{2 \cdot N-2}$, that, in the smallest case of $N = 3$, corresponds to a gain of 25%.

A different attempt to overcome the retiming invariant rule of Equation (1) is given in [2, 3], where the authors propose software pipelining techniques as a better alternative to retiming for sequential circuit optimization. Their goal is to derive an optimum placement of the registers such that the clock period is close (or, in the best case, equal) to the lower-bound of Equation (2). From the optimal schedule found with software pipelining, new registers are placed in the circuit regardless of the number and the position of the original ones. The resulting circuit is a multi-phase clocked circuit, where all clocks have the same period and the phases are automatically determined by the algorithm. Then, edge-triggered flip-flops are used where the combinational delays exactly match that period, whereas level-sensitive latches are used elsewhere, thereby improving the circuit area. In the case of the correlator of Figure 1, the authors can claim to reach the theoretical lower bound of the clock period equal to 10 time units, thereby beating both the best optimally-retimed circuit and the optimally-recycled one. To compare this approach with recycling, the following considerations must be done. First, the two approaches are similar in the sense that they both reduce the circuit critical path by inserting new sequential elements, while making sure that all sequential elements are not *sampled* at every clock iteration. This is achieved *physically* by this approach as different latches are controlled with clock signals with different phases, while recycling does it *virtually* by letting all flip-flops be sampled at every clock iteration to discard then those sampled data coinciding with a $\tau$ symbol. Secondly, this approach gives a faster circuit with simpler components (edge-triggered FFs and level-sensitive latches instead of shells and relay stations) at the price of the higher complexity of having to deal with a multi-phase clocked circuit. Finally, while the register initialization issue is not discussed in [2, 3], it is clear that, to guarantee functional equivalence, the multi-phase circuit must operate according to a pre-defined schedule which makes sure that no spurious data are ever sampled by/from a latch. Conversely, as it is also the case with respect to slowdown, the attractiveness of the recycling alternative is that the underlying latency-insensitive protocol implicitly and automatically encodes the scheduling logic.

## 7 Conclusions

We applied the ideas of recycling and latency-insensitive protocols, which were originally proposed for system-level design, to the timing optimization of synchronous circuits. We showed how recycling can be combined with retiming to get circuit speed-ups that are not achievable by using stand-alone retiming or $c$-slow retiming. We provided an analytical formulation to detect when recycling is advantageous and determine the size of the reachable speed-up. Finally, we compared our approach to the application of software pipelining techniques to sequential circuit optimization.

## References


[1] M. Bohr. Interconnect Scaling - The Real Limiter to High Performance ULSI. *IEEE International Electron Devices Meeting*, pages 241–244, Dec. 1995.

[2] F.-R. Boyer, E. M. Aboulhamid, Y. Savaria, and I.-E. Bennour. Optimal design of synchronous circuits using software pipelining techniques. In *Proc. Intl. Conf. on Computer Design*, pages 62–67, 1998.

[3] F.-R. Boyer, E. M. Aboulhamid, Y. Savaria, and M. Boyer. Optimal design of synchronous circuits using software pipelining techniques. *ACM Trans. on Design Automation of Electronic Systems*, 6(4), 2001.

[4] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, 1998.

[5] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, Sept. 2001.

[6] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Proc. of the Design Automation Conf.*, pages 361–367. IEEE, June 2000.

[7] L. F. Chao and E. H. M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, 1997.

[8] J. Cong and C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. In *Proc. of the Design Automation Conf.*, pages 330–335, June 1998.

[9] C. E. Leiserson. *Area-Efficient VLSI Computation*. PhD thesis, Massachusetts Institute of Technology, 1983. MIT Press.

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

[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*, 6:5–35, 1991.

[13] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 8(9):37–39, Sept. 1997.

[14] M. C. Papaefthymiou. Understanding retiming through maximum average-weight cycles. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 338–348, 1991.

[15] N. Shenoy. Retiming: Theory and Practice. *Integration, the VLSI Journal*, 22:1–21, 1997.

[16] N. Shenoy and R. Rudell. Efficient Implementation of Retiming. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 226–233, Nov. 1994.

[17] G. Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proc. Intl. Conf. Symp. on FPGAs*, pages 177–186. ACM, Feb. 2002.

[18] D. Sylvester and K. Keutzer. Impact of Small Process Geometries on Microarchitecture in System on A Chip. *Proceedings of the IEEE*, 89(4):467–489, Apr. 2001.

[19] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek. Post-Placement C-slow Retiming for the Xilinx Virtex FPGA. In *Proc. Intl. Conf. Symp. on FPGAs*, pages 177–186. ACM, Feb. 2003.