Optimizing Sequential Cycles through Shannon Decomposition and Retiming

Cristian Soviani  Olivier Tardieu  Stephen A. Edwards*
Department of Computer Science, Columbia University, New York

Abstract

Optimizing sequential cycles is essential for many types of high-performance circuits, such as pipelines for packet processing. Retiming is a powerful technique for speeding pipelines, but it is stymied by tight sequential cycles. Designers usually attack such cycles by manually combining Shannon decomposition with retiming—effectively a form of speculation—but such manual decomposition is error-prone.

We propose an efficient algorithm that simultaneously applies Shannon decomposition and retiming to optimize circuits with tight sequential cycles. While the algorithm is only able to improve certain circuits (roughly half of the benchmarks we tried), the performance increase can be dramatic (7%–61%) with only a modest increase in area (3%–12%). The algorithm is also fast, making it a practical addition to a synthesis flow.

1 Introduction

High-performance circuits rely on efficient pipelines. Provided additional latency is acceptable, tight sequential cycles are the main limit to pipeline performance. Unfortunately, such cycles are fundamental to the function of many pipelines.

Retiming [4] is usually applied to high-performance pipelines. Doing so renders the length of purely combinational paths nearly irrelevant since it can divide such paths among multiple clock cycles to increase the clock rate. However, because retiming cannot change the number of registers on a sequential cycle—a loop that passes through combinational logic and one or more registers—the depth of the combinational logic along sequential cycles becomes the bottleneck.

We present an algorithm that uses Shannon decomposition to move combinational logic from one loop to another, making retiming even better for optimizing pipelined sequential circuits. Designers have done this by hand for years; our algorithm applies Shannon decomposition and retiming across an entire circuit to deal with subtle interactions among loops. It considers many implementations at once and selects the best.

Our algorithm works well on pipelined circuits where cycles in control logic are the performance bottlenecks, a typical situation since datapaths rarely include cycles.

---

*soviani, tardieu, sedwards@cs.columbia.edu. Edwards and his group is supported by an NSF CAREER award, a grant from Intel corporation, an award from the SRC, and from New York State’s NYSTAR program.

Figure 1: The single-cycle feedback loop prevents retiming from improving circuit (a), but applying Shannon decomposition (b) reduces the delay around the loop so that (c) retiming can distribute registers and reduce the clock period.

1.1 An example

In the sequential circuit in Figure 1a, combinational block \( f \) has delay \( d_{\text{node}} = 8 \) so the minimum period of this circuit is 8.

The designer put three registers on each input hoping that retiming would distribute them uniformly throughout \( f \) to decrease the clock period. Unfortunately, the feedback loop from the output of \( f \) to its input prevents retiming from improving the period below the combinational length of the loop, \( d_{\text{node}} \), since retiming cannot change the number of registers along it.

Applying Shannon decomposition to this circuit can enable retiming. Figure 1b illustrates how: we have duplicated the combinational logic block and added a multiplexer to its outputs. While this actually increased the longest combinational path to \( d_{\text{node}} + d_{\text{mux}} = 9 \) (we assumed a unit delay for the multiplexer), it greatly reduced the combinational delay around the cycle to the delay of only the mux: \( d_{\text{mux}} \). This transformation makes it possible for retiming to pipeline the slow combinational block to produce the circuit in Figure 1c with a period of \( (1/4)(d_{\text{node}} + d_{\text{mux}}) = 2.25 \).

---

\( f \) = \begin{align*}
x_1 & = x_2 \\
x_2 & = x_3 \\
x_3 & = x_4 \\
x_4 & = y \end{align*}
1.2 Related work

Most sequential optimizations apply a sequence of combinational and sequential transforms that usually interact in unpredictable ways, so most scripts use an empirically-chosen order. By contrast, our work considers the effect of retiming while doing Shannon restructuring, giving better results than either alone. Omitting the retiming step gives an optimization that is easily beaten by a combination of other techniques.

Performance-driven combinational resynthesis is a mature field. Singh et al.’s tree height reduction [12] is typical: it optimizes critical combinational paths at the expense of non-critical ones. Along similar lines, Berman et al. [1] propose the generalized select transform (GST). Like us, the GST employs Shannon decomposition, but our technique also considers the effect of retiming. Other techniques include McGregor’s generalized bypass transform (GBX) [7], which takes advantage of certain types of false paths, and Saldanha’s exact sensitization of critical paths [9], which makes corrections for input patterns that generate a late output.

Our algorithm employs Leiserson and Saxe’s Retiming [4], which can decrease the minimum period of a sequential network by repositioning registers. This commonly-used transformation cannot change the number of registers on a loop; our work employs Shannon decomposition to work around this.

Sequential logic resynthesis has also attracted extensive attention, such as the work of Singh [11]. Malik et al. [5] combine retiming and resynthesis (R&R). Pan [8] proposes a general performance-driven approach to R&R, which our work is a particular instance of. However, we only consider Shannon decomposition instead of arbitrary restructuring, allowing us to systematically explore the design space.

Hassoun et al.’s [3] architectural retiming mixes retiming with speculation and prediction to optimize pipelines; Marinescu et al.‘s technique [6] proposes using stalling and forwarding. Like us, they identify critical cycles as a major performance issue, but they synthesize from high-level specifications and can make architectural decisions. Our work trades this flexibility for more detailed optimizations.

2 Basics

Our algorithm attempts to optimize the speed of sequential circuits that consist of combinational nodes and registers. Formally, a sequential circuit is a directed graph \( S = (V, E) \) with vertices \( V = PI \cup PO \cup N \cup R \cup \{\text{spi}, \text{spo}\} \). \( PI/PO \) are the primary inputs/outputs; \( N \) are single-output combinational nodes; \( R \) are the registers; \( \text{spi} \) and \( \text{spo} \) are two super-nodes connected to/from all \( PI/PO \) respectively. The edges \( E \subseteq V \times V \) model the interconnect: \( \text{fanin}(n) = \{n’ | (n’, n) \in E\} \). \( S \) has no combinational cycles. We define weights \( d : V \rightarrow \mathbb{R} \):

\[
d(n) = \begin{cases} 
\text{arrival time (from clock)} & n \in PI \\
\text{delay of logic} & n \in N \\
\text{required time (to clock)} & n \in PO \\
0 & n \in R \cup \{\text{spi}, \text{spo}\}
\end{cases}
\]

This Boolean property, due to Shannon, has an immediate consequence: if a node is modified as in Figure 2, its computed function \( f \) does not change. This is known as the Shannon or generalized select transform [1].

Our algorithm relies on the fact that the arrival time \( at(n) \) may decrease if \( x_k \) arrives later than all other \( x_i \) (\( i \neq k \)):

\[
at(n) = \max \left\{ \text{at}(f_{\text{SPI}}), \text{at}(f_{\text{SPO}}), \text{at}(x_k) \right\} + d_{\text{max}}
\]

Of course, since both \( f_{\text{SPI}} \) and \( f_{\text{SPO}} \) are computed, the area typically increases. Intuitively, this is speculation, as we start computing \( f \) before knowing \( x_k \).

2.1 Shannon decomposition

Let \( f : \mathbb{B}^p \rightarrow \mathbb{B} \) be the Boolean function of a combinational node \( n \) and let \( 1 \leq k \leq p \). Then

\[
f(x_1, x_2, \ldots, x_p) = x_k f_{x_k} + x_k f_{\overline{x_k}} \quad \text{where}
\]

\[
f_{x_k} = f(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_p) \quad \text{and}
\]

\[
f_{\overline{x_k}} = f(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_p).
\]

Figure 2: Shannon decomposition of \( f \) with respect to \( x_k \)

2.2 Retiming

Retiming [4] follows from noting that moving registers across combinational nodes preserves the circuit functionality. Retiming tries to move registers to decrease long (critical) combinational paths at the expense of short (non-critical) ones. However, it can not decrease the total delay along a cycle.

Let \( \text{ret}(S) \) be the minimum period achievable through retiming. If \( d_{\text{SPI}} \) and \( r_{\text{SPI}} \) are the combinational delay and the number of registers of the cycle \( \mathcal{C} \) in \( S \) then \( \text{ret}(S) \geq d_{\text{SPI}} / r_{\text{SPI}} \). Similarly, if \( \mathcal{P} \) is a path from \( \text{spi} \) to \( \text{spo} \) having \( r_{\mathcal{P}} \) registers and of combinational delay \( d_{\mathcal{P}} \) then \( \text{ret}(S) \geq d_{\mathcal{P}} / (r_{\mathcal{P}} + 1) \). Thus, \( \text{ret}(S) \geq \text{lb}(S) \), where

\[
\text{lb}(S) = \max \left( \max_{\mathcal{C} \in \text{cycles}(S)} \frac{d_{\mathcal{C}}}{r_{\mathcal{C}}}, \max_{\mathcal{P} \in \text{paths}(S, \text{spi}, \text{spo})} \frac{d_{\mathcal{P}}}{r_{\mathcal{P}} + 1} \right)
\]

is known as the fundamental limit of retiming.

Classical retiming may not achieve \( \text{lb}(S) \). To achieve it in general, we must allow registers to be inserted at precise points.
We do not know a priori if applying a Shannon transform to a node improves circuit performance, thus we consider both leaving a node unchanged (Figure 5a) and applying Shannon decomposition to one of its inputs (Figure 5b). We use this to transform single nodes, but we also want to transform groups.

We use a triplet of wires, \((f'_{\pi}, f_{x_{k}}, x_{k})\), essentially a redundant encoding of \(f\) in three bits, to carry a Shannon decomposition between adjacent nodes.

A transformed node, minus the multiplexer, (Figure 5c), has three outputs instead of one: it still computes \(f\), but the output is encoded as above. So \(f\) is transmitted to its fanout(s) by three wires. The fanout node(s) must also be modified to accept the three-wire-encoded function as one of its inputs. This is shown in Figure 5d.

To model a Shannon transform on two connected nodes, we treat the first node as being “start Shannon” (Figure 5c) and the second being “stop Shannon” (Figure 5d). To extend the transform to more than two nodes, we also modify intermediate nodes to be “extend Shannon.” Each has one input and the output encoded on three wires, as shown in Figure 5e.

Transforming a node to be a “start Shannon” can be done in several ways, one for each input. Thus, we have to specify for such nodes which input is used as the select; the other transform types are unambiguous when their context is considered.

As a node with a triplet as output may have several fanouts, the resulting Shannon transform will not be a path in the general case, but a tree, with a “start Shannon” node as root and several “stop Shannon” nodes as leaves.

Any node can be transformed in several ways, and each fanout can use any of these. For example, we can leave a node unchanged and pass its output to one fanout and at the same time transform the node by “start Shannon” and pass the triple output to a second fanout. Our algorithm always makes all variants available to every fanout; it becomes the fanout’s responsibility to select the best.

We deliberately only allow a single triplet to drive a node to limit the area penalty. While the decomposition is easy for multiple incoming triplets, a pair of triplets requires four copies of the node, three would require eight, and so forth.
wire. For single wires, the arrival time is a single real number; for triplets, we have three real numbers, one for each wire. We denote the arrival time of such a triplet as a vector \((a, b, c)\).

For an acyclic fragment with a labeling and arrival times at inputs, we can compute the arrival time at each intermediate wire. For single wires, the arrival time is a single real number; for triplets, we have three real numbers, one for each wire. We denote the arrival time of such a triplet as a vector \((a, b, c)\).

3.2 Shannon decomposition as labeling

We model the result of combining several Shannon transforms in a simple way: we describe it as a node labeling. We consider replacing each combinational node in the network by one or more of the cells in Figure 5, i.e. we label it with letters a–e. For nodes labeled b or c we also have to designate one of its inputs as select, which we do as part of the labeling process.

Some cells have wire triplets as inputs or outputs. Only triplet inputs and outputs can be connected, so not all labelings are valid. But if we respect this simple rule, any labeling of the initial graph is a coherent combination of Shannon transforms. In Figure 6, we choose to perform two Shannon transforms, one involving a single node, the other involving two. Labeling the nodes accordingly, we obtain the circuit in Figure 7.

We can imagine more complex cells, e.g., arising from a multi-input decomposition or nesting transformations. We have experimented with some of these; a thorough investigation remains future work.

3.3 Sets of feasible arrival times and pruning

For an acyclic fragment with a labeling and arrival times at inputs, we can compute the arrival time at each intermediate wire. For single wires, the arrival time is a single real number; for triplets, we have three real numbers, one for each wire. We denote the arrival time of such a triplet as a vector \((a, b, c)\).

Considering all possible valid labelings, we have a set of arrival times for each node \(n\). We call such a set the “feasible arrival times” of \(n\). The set can be a mixture of single numbers and triplets, as different labelings of \(n\) may be replaced by any cell in Figure 5.

Figure 8 illustrates the procedure for computing the FAT set at node \(i\). It is an exhaustive enumeration: the delay of the cell in Figure 5.

FAT sets can be pruned without compromising the performance of the final circuit by keeping only the fastest implementations. An arrival time of \((10,10,14)\) at some node is not necessarily better than an arrival time of \((16)\) since the node may not lie on a critical path. However, \((15)\) is no worse, which we indicate by writing \((15) \leq (16)\). Thus, \((16)\) can be safely removed from a FAT set.

Figures 6 and 7 illustrate the procedure for computing the FAT set at node \(i\). It is an exhaustive enumeration: the delay of the final circuit by keeping only the fastest implementations. An arrival time of \((10,10,14)\) at some node is not necessarily better than an arrival time of \((16)\) since the node may not lie on a critical path. However, \((15)\) is no worse, which we indicate by writing \((15) \leq (16)\). Thus, \((16)\) can be safely removed from a FAT set. The reasoning is more subtle for triplets since the node may not lie on a critical path. However, \((15)\) is no worse, which we indicate by writing \((15) \leq (16)\). Thus, \((16)\) can be safely removed from a FAT set.
3.4 Simultaneously considering several circuits

FAT sets allow us to consider multiple circuit implementations simultaneously. Each fanin has a FAT set that completely characterizes all possible implementations of that fanin.

Fortunately, FAT sets are small enough for us to exhaustively consider, for each node, all possible cells in Figure 5 as well as the two types of each fanin (simple wire or triplet).

To compute the FAT set of \( n \), we consider all choices of node, compute their arrival times, and prune the resulting set. Using this operation instead of (2) in the Bellman-Ford relaxation step (Figure 3) allows us to compute the arrival times for a set of circuit implementations rather than just a single one. Pan [8] uses a similar technique.

We claim that retiming for period \( c \) is feasible iff Bellman-Ford converges. We prove half of this claim by construction. If Bellman-Ford converges, we build an equivalent circuit for which \( \text{lb}(S) \leq c \), so, after retiming, \( c \) is feasible. Such a brute-force construction produces overly large circuits; instead, we use a construction that limits Shannon-induced duplication to critical paths only; see Section 3.5.

Convergence of our augmented Bellman-Ford algorithm implies a fixed-point solution, i.e., a FAT set for each node, which is stable under the pruned FAT-set computation. For the sample in Figure 9a, Bellman-Ford converges to the fixed-point solution in Figure 9b, so we claim the period \( c = 3 \) is feasible.

For each node, we build an implementation corresponding to each element of its FAT set; we are free to choose any cell from Figure 5 and use any FAT elements at each input.

For example, for node \( h \) in Figure 9b, we consider two implementations with FATs of \((4, 4, 8)\) and \((9)\). These are “Start Shannon” and “Shannon” (Figure 5c and b), both with \( g \)’s output as the select. These give arrival times of \((4, 4, 8)\) and \((9)\).

The procedure will succeed for each node as a consequence of how we computed the pruned FAT sets through the Bellman-Ford relaxation. For the resulting network \( \text{lb}(S) \leq c \), so, after retiming, we have a solution.

3.5 Area-oriented construction

Nodes not along critical cycles can use smaller, slower implementations. This basic observation leads to our area-efficient restructuring from FAT sets.

We construct the circuit through a reverse graph traversal. The required time of spo is \( c \). Then, at each node, we select an implementation and propagate required times toward its fanins. Like the arrival times, a required time is either a real number or a triplet (Section 3.3).

At each node, we have a list of one or more required times from each of its fanouts. Using the already-computed FAT sets, we can determine which cells in Figure 5 are feasible for the node for each required time. We build a feasibility table with cells as lines and required times as columns. Each cell has a cost that models its expected area. We select a minimum-cost set of lines that cover all columns.

Figure 9: (a) A circuit with desired period \( c = 3 \); arrival times \( A = 1, B = 3, C = 2 \); required times \( D = 3 \); \( d_{\text{max}} = 1 \). Our extended algorithm computes the FAT sets in (b), implying the restructured circuit in (c). Finally, retiming moves latches (d).
We implemented our algorithm in C++ using the SIS libraries [10] to handle BLIF files. Our testing platform is a 2.5 GHz, 512 MB Pentium 4 running Fedora Core 3 Linux.

We ran our algorithm on mid-sized ISCAS89 sequential benchmarks and target an FPGA-like, 3-input lookup-table architecture. Hence, we report delay as levels of logic and area as the number of lookup tables. SIS failed to run on the other ISCAS89 benchmarks; we do not report their numbers.

Following Saldanha et al. [9], we run script.rugged and perform a speed-oriented decomposition decomp -g; eliminate -1; sweep; speed up -i on each sample. We then reduce the network’s depth while keeping its nodes 3-feasible with reduce depth -f 3 [13]. We report the results of this classical FPGA delay-oriented flow under “Reference” in Table 1.

Starting from these optimized circuits, we compare directly running retiming (retime -n -i, modified to use the unit delay model) with running our algorithm followed by retiming. Columns “Retimed” and “Ours” list the period and area results. Our running time, listed in the “time” column, includes finding the period by binary search. We verified the sequential equivalence of the input and output of our algorithm using VIS [2]; our reported times do not include this.

Although our algorithm can do nothing on half the examples, it gives a significant speed-up for the other half at the expense of an average 5% area increase. The algorithm is very fast, especially when no improvement can be made. Its runtime appears linear in the circuit size. Its memory requirements are low, e.g., 70MB for the largest example s38417. Our technique therefore appears to scale well.

## 5 Conclusions

We presented an algorithm that systematically explores combinations of retiming and Shannon decomposition. Our decompositions are a form of speculation that duplicates logic in general, but we deliberately restrict each node to be duplicated no more than once, bounding the area increase and also simplifying the optimization procedure.

The algorithm finds the optimum-period solution. Our resynthesis technique attempts to limit duplication off the critical path to further limit the area penalty.

Experimental results already show significant speed improvements at the expense of very little area increase. Running times suggest the algorithm scales well to large circuits.