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

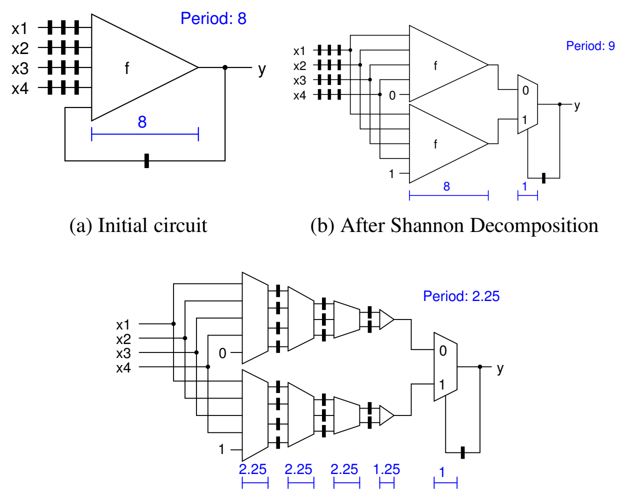
High-performance circuits rely on efficient pipelines. Provided additional latency is acceptable, tight sequential cycles are the main limit to pipeline performance. Unfortunately, such cycles are fundamental to the function of many pipelines.

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.



(a) Initial circuit (b) After Shannon Decomposition

(c) After Shannon Decomposition and Retiming

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_{node}, since retiming can not 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_{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 $(1/4)(d_{\text{node}} + d_{\text{mux}}) = 2.25$.

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 McGeer'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; spi and spo are two super-nodes connected to/from all *PI*/*PO* respectively. The edges $E \subset V \times V$ model the interconnect: $\text{fanin}(n) = \{n' | (n', n) \in E\}$. S has no combinational cycles. We define weights $d : V \to \mathbb{R}$:

$$d(n) = \begin{cases} \text{arrival time (from clock)} & n \in PI \\ \text{delay of logic} & n \in N \\ \text{required time (to clock)} & n \in PO \\ 0 & n \in R \cup \{\text{spi}, \text{spo}\} \end{cases} \tag{1}$$

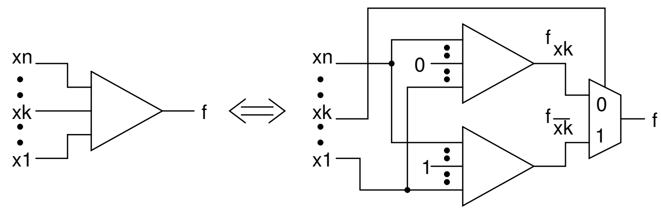


Figure 2: Shannon decomposition of f with respect to x_k

Arrival times are computed in a topological order on the combinational nodes:

$$\text{at}(n) = d(n) + \max_{n' \in \text{fanin}(n)} \text{at}(n') \tag{2}$$

2.1 Shannon decomposition

Let $f : \mathbb{B}^p \to \mathbb{B}$ be the Boolean function of a combinational node n and let $1 \le k \le p$. Then

$$\begin{aligned} f(x_1, x_2, \ldots, x_p) &= x_k f_{x_k} + \overline{x_k} f_{\overline{x_k}} \quad \text{where} \\ f_{x_k} &= f(x_1, \ldots, x_{k-1}, 1, x_{k+1}, \ldots, x_p) \quad \text{and} \\ f_{\overline{x_k}} &= f(x_1, \ldots, x_{k-1}, 0, x_{k+1}, \ldots, x_p). \end{aligned}$$

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 $\text{at}(n)$ may decrease if x_k arrives later than all other x_i $(i \ne k)$:

$$\text{at}(n) = \max\left\{\text{at}(f_{\overline{x_k}}), \text{at}(f_{x_k}), \text{at}(x_k)\right\} + d_{\text{mux}}$$

Of course, since both $f_{\overline{x_k}}$ and f_{x_k} are computed, the area typically increases. Intuitively, this is speculation, as we start computing f before knowing 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 $d_{\mathscr{C}}$ and $r_{\mathscr{C}}$ are the combinational delay and the number of registers of the cycle $\mathscr{C}$ in S then $\text{ret}(S) \ge d_{\mathscr{C}}/r_{\mathscr{C}}$. Similarly, if $\mathscr{P}$ is a path from spi to spo having $r_{\mathscr{P}}$ registers and of combinational delay $d_{\mathscr{P}}$ then $ret(S) \ge d_{\mathscr{P}}/(r_{\mathscr{P}}+1)$. Thus, $\text{ret}(S) \ge \text{lb}(S)$ where

$$\text{lb}(S) = \max\left(\max_{\mathscr{C} \in \text{cycles}(S)} \frac{d_{\mathscr{C}}}{r_{\mathscr{C}}}, \max_{\mathscr{P} \in \text{paths}(S, \text{spi}, \text{spo})} \frac{d_{\mathscr{P}}}{r_{\mathscr{P}}+1}\right) \tag{3}$$

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

```
procedure RelaxNode(n)
    at_new ← d(n) + max_{n′∈fanin(n)} at(n′)
    if at_new ≠ at(n) then
        at(n) ← at_new
```

Figure 3: The Bellman-Ford relaxation step

```
procedure SeqShannon(S, c)
    (converges, fix_point_fat) = Bellman-Ford (S, c)
    if not converges then
        return NOT_FEASIBLE
    ShannonTransform(S, c, fix_point_fat)
    Retime(S)
    return SUCCESS
```

Figure 4: SeqShannon: Our algorithm for restructuring a circuit S to achieve a period c

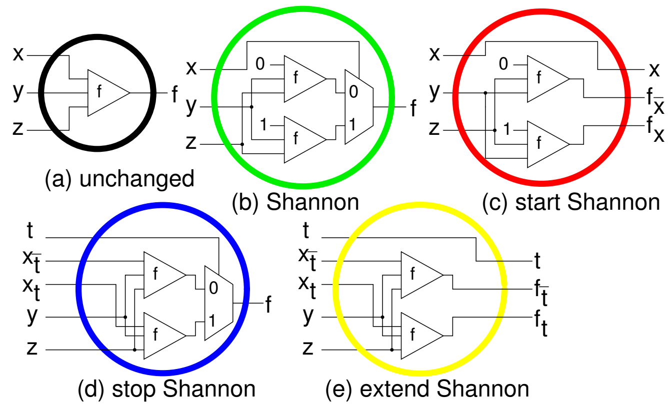


Figure 5: The Shannon “cell library”

inside the nodes. We will assume this is possible (which it is, for example, in FPGAs [14]), so $\mathrm{ret}(S) = \mathrm{lb}(S)$ holds. We shall thus focus on transforming S to minimize $\mathrm{lb}(S)$.

Computing $\mathrm{lb}(S)$ directly using equation (3) is not practical because the number of cycles may be exponential; instead we use the Bellman-Ford minimum-length-path algorithm. To apply Bellman-Ford, which has no notion of registers, we treat the registers as nodes with negative delay: $\forall r \in R, d(r) = -c$, where c is the desired period. This trick follows from noticing that for each $\mathscr{C} \in \mathrm{cycles}(S)$, $c \geq \mathrm{lb}(S)$, so $c \geq d_{\mathscr{C}}/r_{\mathscr{C}}$. Rewriting, we have $d_{\mathscr{C}} - c \cdot r_{\mathscr{C}} \leq 0$, which implies $\sum_{n\in\mathscr{C}} d(n) \leq 0$.

If there exists a retiming for period c, then S has no positive cycles, and, due to a similar reasoning for paths $\mathscr{P}$, $\mathrm{at}(spo) \leq c$. The reverse implication, which is harder to prove, also holds.

These conditions hold iff the Bellman-Ford algorithm ends in a bounded number of iterations. Therefore, $\mathrm{lb}(S)$ can be approximated by binary search on the period c. Note that the Bellman-Ford relaxation step (Figure 3) is based on (2).

3 Our algorithm: combining Shannon and retiming

Our algorithm (Figure 4) systematically explores combinations of Shannon transforms and retiming for a given period c. If feasible, it returns a circuit operating with period c, and fails otherwise. We can find the best c by binary search.

The core of our technique is a modified Bellman-Ford shortest-path algorithm. Given a period c, it looks for a way to reshape the combinational logic such that $c \geq \mathrm{lb}(S)$, where $\mathrm{lb}(S)$ is defined by (3). If it succeeds, we are guaranteed that ShannonTransform() and Retime() will be able to restructure the circuit and achieve the desired period c. It is fairly easy to build a correct circuit at this point, but harder to build one with reasonable area. We discuss doing this in Section 3.5.

3.1 Chaining Shannon transforms

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, $(f_{\overline{x_k}}, f_{x_k}, x_k)$, 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.

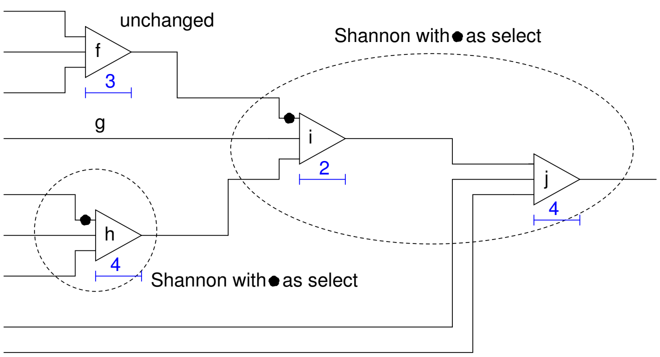
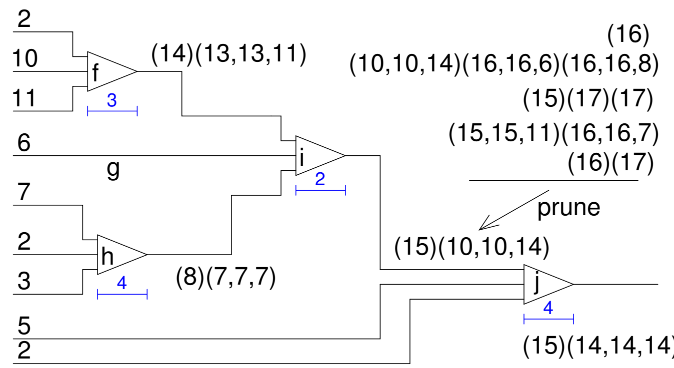


Figure 6: A circuit to illustrate Shannon decomposition



	(none)	selector f	selector g	selector h
unchanged	(16)			
start		(10,10,14)	(16,16,6)	(16,16,8)
start & stop		(15)	(17)	(17)
extend		(15,15,11)		(16,16,7)
extend & stop		(16)		(17)

Figure 8: Computing feasible arrival times for node i

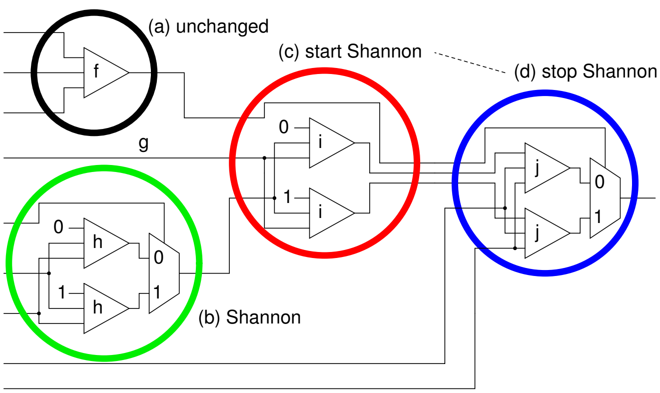


Figure 7: The circuit in Figure 6 transformed with Shannon decomposition

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 input 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 $(\mathrm{at}(f_{\overline{x_k}}), \mathrm{at}(f_{x_k}), \mathrm{at}(x_k))$.

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" or $\mathrm{fat}(n)$. 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 "unchanged" case is calculated first, then multiple possibilities are calculated assuming each input can be a selector: starting Shannon with the input as a selector, starting and immediately stopping, extending Shannon if the input has at least one triplet in its pruned FAT, and extending Shannon and stopping.

FAT sets can be pruned without compromising the performance of the final circuit by keeping only the fastest implementations. An arrival time of (15) 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 $(15) \preceq (16)$. Thus, (16) can be safely removed from a FAT set. The reasoning is more subtle for triplets of wires, but similar: $(16,16,6) \preceq (16,16,8)$, so $(16,16,8)$ can also be removed. In general, $(a) \preceq (x,y,z)$ if $a \leq x$ and $(a,b,c) \preceq (x,y,z)$ if $a \leq x$, $b \leq y$, and $c \leq z$. In Figure 8, exhaustively applying this rule to the FAT set for node i gives only two elements: $\mathrm{fat}(i) = \{(15), (10,10,14)\}$.

The small size of the FAT sets after pruning in Figure 8 is typical; across all the circuits we have analyzed, we find pruned FAT sets seldom contain more than four elements. This is probably the main reason that our algorithm is fast.

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 $\mathrm{lb}(S) \leq c$, 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).

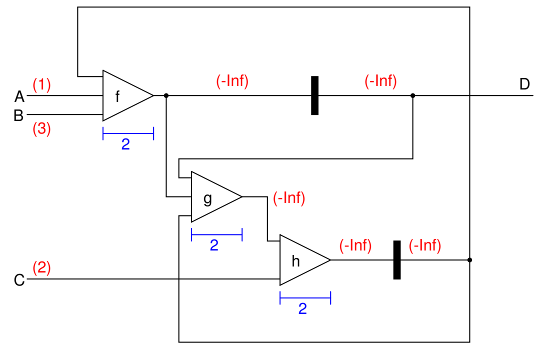
The procedure will succeed for each node as a consequence of how we computed the pruned FAT sets through the Bellman-Ford relaxation. For the resulting network $\mathrm{lb}(S) \leq c$, so, after retiming, we have a solution.

3.5 *Area-oriented construction*

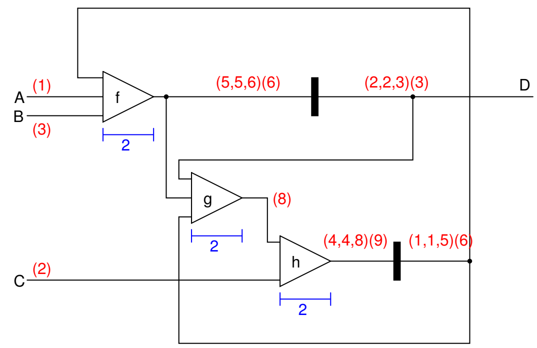
Nodes not along critical cycles can use smaller, slower implementations. This basic observation leads to our area-efficient restructuring from FAT sets.

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).

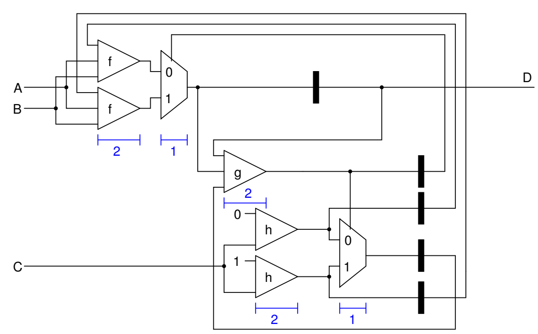
At each node, we have a list of one or more required times from each of its fanouts. Using the already-computed FAT sets, we can determine which cells in Figure 5 are feasible for the node for each required time. We build a feasibility table with cells as lines and required times as columns. Each cell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



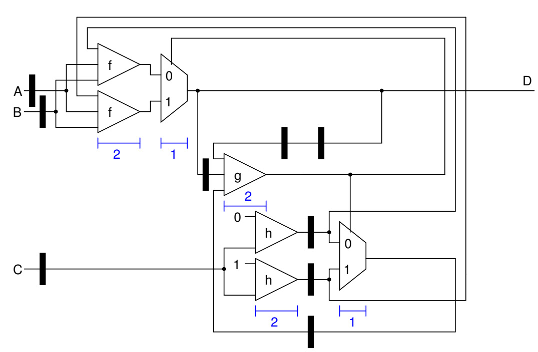
(a) Initial circuit



(b) Fixed-point solution



(c) After Shannon decomposition



(d) Final retimed circuit

Figure 9: (a) A circuit with desired period $c = 3$; arrival times $A = 1, B = 3, C = 2$; required times $D = 3$; $d_{\mathrm{mux}} = 1$. Our extended algorithm computes the FAT sets in (b), implying the restructured circuit in (c). Finally, retiming moves latches (d).

Table 1: Experiments on ISCAS89 sequential benchmarks

	Reference		Retimed		Ours		Time	Speed	Area
	period	area	period	area	period	area	(s)	up	penalty
s510	8	184	8	184	8	184	0.5		
s641	11	115	11	115	9	122	1.1	22%	6%
s713	11	118	11	118	10	121	0.9	10%	3%
s820	7	206	7	206	7	206	0.5		
s832	7	217	7	217	7	217	0.4		
s838	10	154	10	154	8	162	2.6	25%	5%
s1196	9	365	9	365	9	365	0.6		
s1423	24	408	21	408	13	460	3.8	61%	12%
s1488	6	453	6	453	6	453	0.7		
s1494	6	456	6	456	6	456	0.8		
s9234	11	662	8	656	8	684	6.7		
s13207	14	1382	11	1356	9	1416	18.0	22%	4%
s38417	14	7706	14	7652	13	7871	113	7%	3%

Although our procedure usually selects one implementation of each node, this is not always the case. For example, node h in Figure 9d is both "end Shannon" (the output from the multiplexer fed to node g) and "extend Shannon" (the outputs before the multiplexer fed to f). Note that our algorithm does not needlessly duplicate node h in this case.

The area cost estimate for each type of node implementation is a heuristic; we are in the process of refining them.

Starting with the fixed-point solution in Figure 9b, we generate the network in Figure 9c. Its minimum period is 9, which is worse than the original in Figure 9a, but $\mathrm{lb}(S)$ as defined by (3) is 3. Retiming gives the network in Figure 9d, which behaves like Figure 9a but respects the timing constraint.

4 Experiments

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.

[1] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. Trevillyan. Efficient techniques for timing correction. In *Proc. ISCAS*, pages 415–419, 1990.

[2] R. K. Brayton et al. VIS: a system for verification and synthesis. In *Proc. Computer-Aided Verification*, pages 428–432, 1996.

[3] S. Hassoun and C. Ebeling. Architectural retiming: pipelining latency-constrained circuits. In *DAC*, pages 708–713, 1996.

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

[5] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential networks with combinational techniques. *IEEE Transactions on CAD*, 10(1):74–84, 1991.

[6] M.-C. V. Marinescu and M. Rinard. High-level automatic pipelining for sequential circuits. In *ISSS*, pages 215–220, 2001.

[7] P. C. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. K. Sahni. Performance enhancement through the generalized bypass transform. In *ICCAD*, pages 184–187, 1991.

[8] P. Pan. Performance-driven integration of retiming and resynthesis. In *Proceedings of DAC*, pages 243–246, 1999.

[9] A. Saldanha, H. Harkness, P. C. McGeer, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Performance optimization using exact sensitization. In *Proc. DAC*, pages 425–429, 1994.

[10] E. M. Sentovich et al. SIS: A system for sequential circuit synthesis. Technical report, UCB/ERL M92/41, 1992.

[11] K. J. Singh. *Performance optimization of digital circuits*. PhD thesis, University of California, Berkeley, 1992.

[12] K. J. Singh, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proceedings of ICCAD*, pages 282–285, 1988.

[13] H. Touati, H. Savoj, and R. K. Brayton. Delay optimization of combinational logic circuits by clustering and partial collapsing. In *Proceedings of ICCAD*, pages 188–191, 1991.

[14] H. Touati, N. Shenoy, and A. L. Sangiovanni-Vincentelli. Retiming for table-lookup field-programmable gate arrays. In *Proc. Intl. Workshop FPGAs*, pages 89–93, 1992.