Optimizing Sequential Cycles Through Shannon Decomposition and Retiming

Cristian Soviani, Olivier Tardieu, and Stephen A. Edwards, *Senior Member, IEEE*

***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 (1%–12%). The algorithm is also fast, making it a practical addition to a synthesis flow.**

***Index Terms*—Circuit optimization, circuit synthesis, encoding, sequential logic circuits.**

I. Introduction

LOGIC synthesis procedures typically consist of a collection of algorithms applied in sequence that make fairly local modifications to a digital circuit. Usually, these algorithms take small steps through the solution space, i.e., by making many little perturbations of a circuit, and do not take into account what their successors can do to the circuit. Such an approach, while simple to code, often leads to suboptimal circuits.

In this paper, we propose a logic synthesis procedure that considers a postretiming step while resynthesizing an entire logic circuit using Shannon decomposition to add speculation. The result is an efficient procedure that produces faster circuits than performing Shannon decomposition and retiming in isolation.

Our procedure is most effective on high-performance pipelines. Retiming [1] is usually applied to such circuits, which renders the length of purely combinational paths nearly irrelevant since retiming 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.

Manuscript received April 3, 2006; revised August 9, 2006. This work was supported in part by a National Science Foundation CAREER Award, in part by a grant from Intel Corporation, in part by an award from the Semiconductor Research Corporation (SRC), and in part by the New York State's NYSTAR program. This paper was recommended by Guest Editor D. Sciuto.

The authors are with the Department of Computer Science, Columbia University, New York, NY 10027 USA (e-mail: soviani@cs.columbia.edu; tardieu@cs.columbia.edu; sedwards@cs.columbia.edu).

Digital Object Identifier 10.1109/TCAD.2007.890583

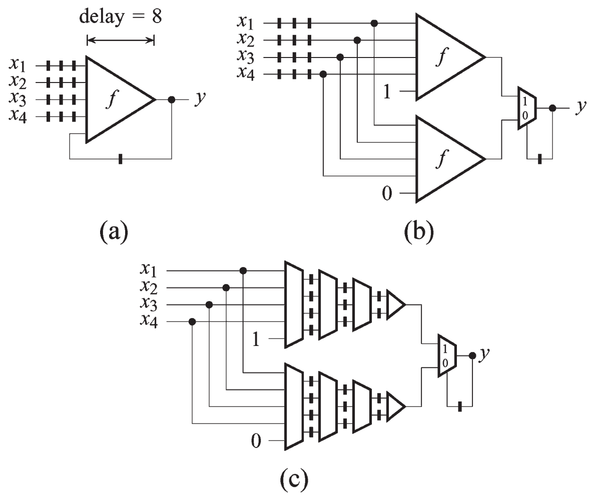


Fig. 1. (a) Single-cycle feedback loop prevents retiming from improving this circuit, but (b) applying Shannon decomposition reduces the delay around the loop so that (c) retiming can distribute registers and reduce the clock period.

Shannon decomposition provides a way to restructure logic to hide the effects of late-arriving signals. This is done by duplicating a cone of logic, feeding constant 1s and 0s into the late-arriving signal and placing a (fast) two-input multiplexer on the output of the two cones.

Following Shannon decomposition with retiming can greatly improve overall circuit performance. Since Shannon decomposition can move logic out of sequential loops, a subsequent retiming step can better balance the logic to reduce the minimum clock period, giving a more efficient circuit.

Combining Shannon decomposition with retiming is a well-known manual design technique, but to our knowledge, ours is the first automated algorithm for it.

A. Example

In the sequential circuit in Fig. 1(a), the combinational block f has delay 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, which is 8, since retiming cannot change the number of registers along it.

```
procedure Restructure($S$,$c$)
    feasible arrival times $\leftarrow$ Bellman-Ford($S$,$c$)
    if feasible arrival time computation failed then
        return FAIL
    Resynthesize($S$,$c$, feasible arrival times)
    Retime($S$)
    return $S$
```

Fig. 2. Our algorithm for restructuring a circuit S to achieve a period c.

Applying Shannon decomposition to this circuit can enable retiming. Fig. 1(b) illustrates how: We have made two duplicates of the combinational logic block and added a multiplexer to their outputs. While this actually increased the longest combinational path to $8 + 1 = 9$ (throughout this paper, we assume that multiplexers have unit delay), it greatly reduced the delay around the cycle to the delay of only the mux, namely one. This enables retiming to pipeline the slow combinational block to produce the circuit in Fig. 1(c), which has a much shorter clock period of $(1/4)(8 + 1) = 2.25$.

The main strength of our algorithm is its ability to consider a later retiming step while judiciously selecting where to perform Shannon decomposition. For example, a decomposition algorithm that did not consider the effect of retiming would reject the transformation in Fig. 1(b) because it made the circuit both larger and slower.

```
function BellmanFord($S$,$c$)
    for each register $n$ in $S$ do
        $d(n) \leftarrow -c$
    for each vertex $n$ in $S$ do
        fat($n$) $\leftarrow \{(-\infty)\}$
    fat(spi) $\leftarrow \{(0)\}$
    for $i = 0$ to max-iterations do
        for each vertex $n$ in $S$ in topological order do
            Relax($n$)
        if there exists $(t)$ in fat(spo) such that $t > c$ then
            return FAIL
        if there were no changes in this iteration then
            return fat
    return FAIL
procedure Relax($n$)
    $F \leftarrow \emptyset$
    for each $p \in$ fanin($n$) do
        $T \leftarrow$ the arrival times of $s_0$-encoded fanins $\neq p$
        for each $t \in$ fat($p$) do
            $s_i \leftarrow$ the encoding of $t$
            for $o = 0$ to $i$ do
                add AT($\langle s_i, s_i, s_o \rangle, t, T, d(n)$) to $F$
                add AT($\langle s_i, s_{i+1}, s_o \rangle, t, T, d(n)$) to $F$
            add AT($\langle s_i, s_{i+1}, s_{i+1} \rangle, t, T, d(n)$) to $F$
    while there are $p, q \in F$ s.t. $p \preceq q$ and $p \neq q$ do
        remove $q$ from $F$
    if $F \neq$ fat($n$) then
        fat($n$) $\leftarrow F$
```

Fig. 3. Our Bellman–Ford algorithm for computing FATs.

B. Overview of the Algorithm

Our algorithm (Fig. 2) takes a network S and a timing constraint (a target clock period) c and uses resynthesis and retiming to produce a circuit with period c if one can be found, or returns failure.

Our algorithm operates in three phases. In the first phase, "Bellman–Ford" (shown in Fig. 3 and described in detail in Section II), we consider all possible Shannon decompositions by considering different ways of restructuring each node. This procedure vaguely resembles technology mapping in that it considers replacing each gate with one taken from a library but does so in an iterative manner because it considers circuits with (sequential) loops. More precisely, the algorithm attempts to compute a set of feasible arrival times (FATs) for each signal in the circuit that indicate that the target clock period c can be achieved after resynthesis and retiming. If the smallest such c is desired, our algorithm is fast enough to be used as a test in a binary search that can approximate the lowest possible c.

In the second phase ("resynthesize," as described in Section III), we use the results of this analysis to resynthesize the combinational gates in the network, which is nontrivial because to conserve area, we wish to avoid the use of the most aggressive (read: area-consuming) circuitry everywhere but on the critical paths. As we saw in the example in Fig. 1, the circuit generated after the second phase usually has worse performance than the original circuit.

We apply classical retiming to the circuit in the third phase, which is guaranteed to produce a circuit with period c.

In Section IV, we present experimental results that suggest that our algorithm is efficient and can produce a substantial speed improvement with a minimal area increase on half of the circuits we tried; our algorithm is unable to improve the other half.

C. Related Work

The spirit of our approach is a fusion of Pan's technique for considering retiming while performing resynthesis [2] with the technology-mapping technique of Lehman *et al.* [3], which implicitly represents many different circuit structures. However, the details of our approach differ greatly. Unlike Pan, we consider a much richer notion of arrival time due to our considering many circuit structures simultaneously, and our resynthesis technique bears only a passing resemblance to classical technology mapping as our "cell library" is implicit and we consider reencoding signals beyond simple inversion.

Performance-driven combinational resynthesis is a mature field. Singh *et al.*'s tree-height reduction [4] is typical: It optimizes critical combinational paths at the expense of noncritical ones. Along similar lines, Berman *et al.* [5] 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 *et al.*'s generalized bypass transform [6], which takes advantage of certain types of false paths, and Saldanha *et al.*'s exact sensitization of critical paths [7], which makes corrections for input patterns that generate a late output.

Our algorithm employs Leiserson and Saxe's retiming [1], 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; this paper employs Shannon decomposition to work around this.

Sequential logic resynthesis has also attracted extensive attention, such as the work of Singh [8]. Malik *et al.* [9] combine retiming and resynthesis (R&R). Pan's approach to R&R [2] is a superset of ours, but our restriction to Shannon decomposition allows us to explore the design space more systematically.

Hassoun and Ebeling's [10] architectural retiming mixes retiming with speculation and prediction to optimize pipelines; Marinescu and Rinard's technique [11] 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. This paper trades this flexibility for more detailed optimizations.

II. First Phase: Computing Arrival Times

To account for retiming, we use a modified Bellman–Ford algorithm (Fig. 3) instead of classical static timing analysis to determine whether the fastest circuit we can find can be retimed so as to achieve period c. If the first phase is successful, it produces as a side-effect arrival times that guide our resynthesis algorithm, which we describe in Section III.

A. Basics

Our algorithm operates on sequential circuits that consist of combinational nodes and registers. Formally, a sequential circuit is a directed graph $S = (V, E)$ with vertices $V = PI \cup PO \cup N \cup R \cup \{\mathrm{spi}, \mathrm{spo}\}$. PI and PO are the primary inputs and outputs, respectively; N are the single-output combinational nodes; R are the registers; and spi and spo are two supernodes connected to and from all PIs and POs, respectively. The edges $E \subset V \times V$ model the interconnect: $\text{fan-in}(n) = \{n' | (n', n) \in E\}$. We assume that S has no combinational cycles and each vertex is on a path from spi to spo. We define weights $d : V \to \mathbb{R}$, which represent the following:

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

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

$$\mathrm{at}(n) = d(n) + \max_{n' \in \text{fan-in}(n)} \mathrm{at}(n'). \quad (1)$$

B. 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,

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

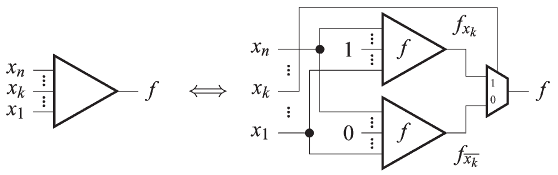


Fig. 4. Shannon decomposition of f with respect to x_k.

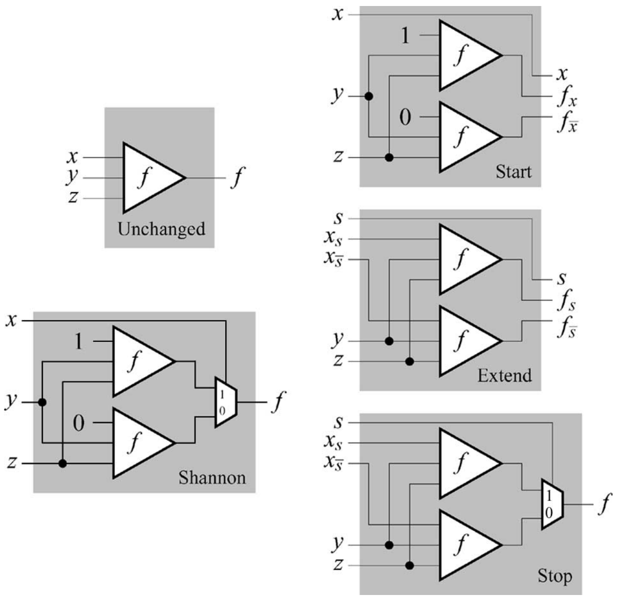


Fig. 5. Basic Shannon "cell library."

This Boolean property, due to Shannon, has an immediate consequence: Modifying a node, as shown in Fig. 4, leaves its function unchanged even though its input-to-output delays change. This is known as the Shannon or GST [5].

Our algorithm relies on the fact that arrival time $\mathrm{at}(n)$ may decrease if x_k arrives later than all other x_i $(i \neq k)$s, i.e.,

$$\mathrm{at}(n) = \max\{\mathrm{at}(f_{\overline{x_k}}), \mathrm{at}(f_{x_k}), \mathrm{at}(x_k)\} + d_{\mathrm{mux}}.$$

Since the circuit must compute both $f_{\overline{x_k}}$ and f_{x_k}, the area typically increases. Intuitively, this is speculation: we start computing f before knowing x_k.

C. Shannon Decomposition as a Kind of Technology Mapping

A key to our technique is our ability to consider many different resynthesis options simultaneously. Our approach resembles technology mapping in that we consider replacing each node in a network with one taken from a cell library. Fig. 5 is the beginnings of our library for characterizing multiple Shannon decompositions (our full algorithm considers a larger library that builds on this one—see Section II-E). *Unchanged* leaves the node unchanged, and *Shannon* bypasses one of the inputs using Shannon decomposition. Both of these are local changes; the *Start* variant begins a Shannon decomposition that can either be extended by *Extend* or terminated by *Stop*.

While the *Unchanged* and *Shannon* cells can be used in arbitrary places, the (three-wire) output of a *Start* cell can only feed the three-wire input of an *Extend* or *Stop* cell. Furthermore,

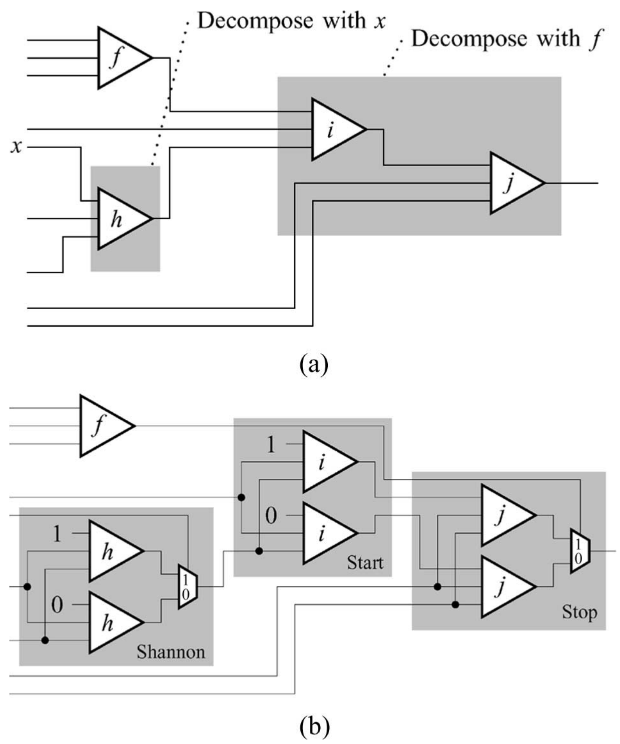


Fig. 6. Shannon decomposition through node replacement. (a) Initial circuit. (b) After Shannon transformation.

to minimize node duplication, we only allow a single Shannon-encoded input per node, so at most one input to an *Extend* node may be *Start* or *Extend*.

Fig. 6 illustrates how this works. Fig. 6(a) shows the two Shannon transforms we wish to perform, with one involving a single node and the other involving two. We replace node h with *Shannon*, node i with *Start*, and node j with *Stop*. Fig. 6(b) shows the resulting circuit, which embodies the transformation we wanted.

D. Redundant Encodings and Arrival Times

Using Shannon decomposition to improve circuit performance is a particular case of the more general idea of using redundant encodings to reduce circuit delay. The main challenge in producing a fast circuit is producing as early as possible the inputs of gates that are nearer the outputs of the circuit. There is not much flexibility when a single bit travels down a single wire, but using multiple wires to transmit a single bit might allow the sender to transmit partial information earlier. This can enhance performance if the downstream computation can be restructured so it performs most of the computation using only partial data and then quickly calculating the final result using the remaining late-arriving data.

Fig. 7 illustrates this idea on the Shannon transformation of a single node. On the left, inputs x, y, and z are each conveyed on a single wire, which is a trivial encoding we label s_0. Just after that, however, we choose to reencode x using the three-wire encoding s_1, in which one wire selects which of the other two wires actually carries the value. This is the basic Shannon-inspired encoding. On the left side of Fig. 7, this

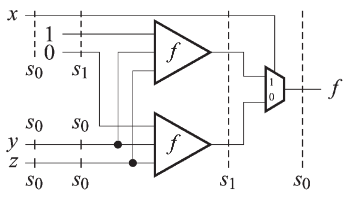


Fig. 7. Shannon transform as redundant encoding.

"encoding" step amounts to adding two constant-value wires and interpreting the original wire x as a select. However, once we feed this encoding into the two copies of f, the result is more complicated. Again, we are using the s_1 encoding with x as the select wire, but instead of two constant values, the two other wires carry f_x and $f_{\overline{x}}$. On the right of Fig. 7, we use a two-input multiplexer to decode the s_1 encoding into the single-wire s_0 encoding.

Using such a redundant encoding will speed up the operation of the circuit if the x signal arrives much later than the y or z signals. When we reencode a signal, the arrival times of its various components can differ, which is the source of the speedup. For example, at the s_1 encoding of x, the x wire arrives later, while the two constant values arrive instantly.

A central trick in the first phase of our algorithm is the observation that only arrival times matter when considering which cell to use for a particular node. In particular, the detailed topology of the circuit, such as whether a Shannon decomposition had been used, is irrelevant when considering how best to resynthesize a node. Only arrival times and the encoding of each arriving signal matter.

E. Our Family of Shannon Encodings

While a general theory of reencoding signals for circuit resynthesis could be (and probably should be) developed, in this paper, we restrict our focus to a set of encodings derived from Shannon decompositions that aim to limit area overhead. In particular, evaluating a function with a single encoded input only ever requires us to make two copies of the original function.

The basic cell library of Fig. 5 works well for most circuits, but some transformations demand the richer library we describe in this section. For example, the larger family is required to convert a ripple-carry adder into a carry-select adder.

Technically, we define an encoding as a function $e : \mathbb{B}^k \to \mathbb{B}$ that maps information encoded on k wires to a single bit. Our family of Shannon-inspired encodings $S = \{s_0, s_1, \ldots\}$ are defined recursively, i.e.,

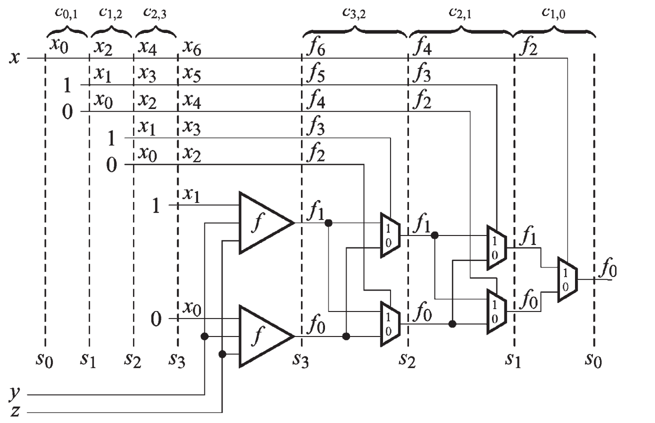
$$s_i = \begin{cases} x_0 \mapsto x_0, & \text{if } i = 0 \\ x_0, \ldots, x_{2i} \mapsto & \\ s_{i-1}(\overline{x_2}x_0 + x_2x_1, \overline{x_3}x_0 + x_3x_1, x_4, \ldots, x_{2i}), & \text{otherwise.} \end{cases}$$

The first few such encodings are listed here

$$s_0 = x_0 \mapsto x_0$$

$$s_1 = x_0, x_1, x_2 \mapsto \overline{x_2}x_0 + x_2x_1$$

$$s_2 = x_0, x_1, x_2, x_3, x_4 \mapsto \overline{x_4}(\overline{x_2}x_0 + x_2x_1) + x_4(\overline{x_3}x_0 + x_3x_1).$$



$c_{0,1}$ $c_{1,2}$ $c_{2,3}$ $c_{3,2}$ $c_{2,1}$ $c_{1,0}$
x
x_0 x_2 x_4 x_6 f_6 f_4 f_2
x_1 x_3 x_5 f_5 f_3
x_0 x_2 x_4 f_4 f_2
x_1 x_3 f_3
x_0 x_2 f_2
f_1 f_0
f
s_0 s_1 s_2 s_3 s_3 s_2 s_1 s_0
y
z

Fig. 8. Encoding x, evaluating $f(x, y, z)$, and decoding the output for the s_0, s_1, s_2, and s_3 codes.

Fig. 8 shows a circuit that takes three inputs x, y, and z; encodes x to the s_1, s_2, and s_3 codes before evaluating $f(x, y, z)$ with the s_3-encoded x; and then decodes the result back to the single-wire s_0 encoding. While this circuit as a whole is worse than the simple Shannon decomposition of Fig. 7, subsets of this circuit provide a way to move between encodings.

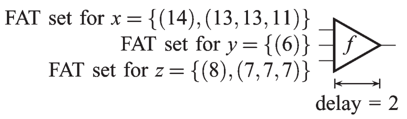
We think of the layers of Fig. 8 as "Shannon codecs"—small circuits that transform an encoding s_a to s_b. We write $c_{a,b}$ for such a codec circuit. For $a < b$, $c_{a,b}$ just adds pairs of wires connected to constant 0s and 1s. For $a > b$, $c_{a,b}$ consists of some multiplexers.

We chose our family of Shannon encodings so that evaluating a functional block with a single input with a higher order encoding only ever requires two copies of the functional block. Fig. 8 shows this for the s_3 case; other cases follow the same pattern.

In general, we only allow a single encoded input to a cell (all others are assumed to be single wires, i.e., s_0-encoded). To evaluate a function f, we make two copies of it, feed the nonencoded signals to each copy, feed x_0 and x_1 from the encoded input signal to the two copies, which compute f_0 and f_1 of the encoded output signal, and pass the rest of x_i from the encoded signal directly through to form f_2, etc. The *Extend* cell in Fig. 5 is a case of this pattern for encoding s_1.

We consider one important modification of this basic idea: the introduction of "codec" stages at the inputs and outputs of a cell. Following the rules suggested in Fig. 8, we could place arbitrary sequences of encoders and decoders on the inputs and output of a functional block, but to limit the search space, we only consider cells that place one encoder on a single input and one or more decoders on the output. For instance, the *Stop* cell in Fig. 5 is derived from the *Extend* cell by appending a decoder to its output. Similarly, the *Start* cell consists in an encoder plus the *Extend* cell. Finally, the *Shannon* cell adds both an encoder and a decoder to the *Extend* cell.

Neither the number of inputs of a cell nor the index of the encoded input matters to the computation of arrival times since we assume that each node has the same delay from each input to the output. Therefore, we identify a cell with a triplet of encodings: $\langle s_i, s_f, s_o \rangle$, where s_f denotes the encoding of the encoded input fed to the combinational logic, s_i denotes the encoding of the input at the interface of the cell (which may differ from s_f because of an encoder: $i = f$ or $i = f - 1$), and s_o denotes the encoding of the output of the cell (which may differ from s_f because of decoders: $0 \leq o \leq f$).



FAT set for $x = \{(14), (13, 13, 11)\}$
FAT set for $y = \{(6)\}$
FAT set for $z = \{(8), (7, 7, 7)\}$
f
delay = 2

	Unchanged	**Shannon**	**Start**	**Stop**	**Extend**
	$\langle s_0,s_0,s_0 \rangle$	$\langle s_0,s_1,s_0 \rangle$	$\langle s_0,s_1,s_1 \rangle$	$\langle s_1,s_1,s_0 \rangle$	$\langle s_1,s_1,s_1 \rangle$
x	(16)	(15)	(10,10,14)	(16)	(15,15,11)
y	(16)	(17)	(16,16,6)		
z	(16)	(17)	(16,16,8)	(17)	(16,16,7)

Fig. 9. Computing FATs for a node f.

The columns of the table in Fig. 9 correspond to the triplets for each of the five cells in the basic Shannon cell library (Fig. 5).

Our algorithm considers many additional cell types, which arise from employing higher degree encodings. In theory, there are an infinite number of such cells, but in practice, only a few more are ever interesting. The circuit in Fig. 8 illustrates some higher degree encodings, corresponding to $\langle s_0, s_3, s_0 \rangle$. Note that our algorithm would never generate this particular cell in its entirety because if s_f is s_3, then s_i may be either s_3 or s_2, not s_0. It could, however, generate the cell with an s_2-encoded input, i.e., $\langle s_2, s_3, s_0 \rangle$.

F. Sets of FATs

The first phase of our algorithm (Fig. 3) attempts to compute a set of FATs that will guide us in resynthesizing the initial circuit into a form that can be retimed to give us the target clock period c.

In classical static timing analysis, the arrival time at the output of a gate is a single number: the maximum of the arrival times at its input plus the intrinsic delay of the gate [see (1)]. In our setting, however, we represent the arrival time of a signal with a set of tuples, each representing the arrival of the signal under a particular encoding. Considering sets of arrival times allows us to simultaneously consider different circuit variants.

Our arrival-time tuples contain one real number per wire in the encoding. Since no two encodings use the same number of wires, the arity of the tuple effectively defines the encoding. For example, an arrival-time three-tuple such as (2, 2, 3) is always associated with encoding s_1 since it is the only one comprised of three wires.

The example in Fig. 9 illustrates how we compute the set of FATs at a node f through brute-force enumeration. The code for this appears in lines 17–24 of Fig. 3.

The FAT sets shown in Fig. 9 indicate that we know input x can arrive as early as time 14 as an unencoded (s_0) signal or at times 13, 13, and 11 for the three wires in an s_1-encoded signal. Similarly, y can arrive as early as time 6, and z may arrive as early as time 8 in s_0 form or (7, 7, 7) in s_1 form.

Considering only the cell library of Fig. 5, the table in Fig. 9 shows how we enumerate the possible ways f can be implemented. The rows of this table correspond to the iteration over the fan-ins of the node—line 17 in Fig. 3. The columns are produced from the iterations over input encoding from the fan-in (line 19) and output encoding (line 21).

The *Unchanged* case is considered first for each input. Since every input has the s_0 encoding for this case, we obtain the same arrival time for each input. Here, this is $14 + 2 = 16$, which is the earliest arrival time of x, which is the latest arriving s_0 signal, plus 2, i.e., the delay of f.

The *Shannon* case is considered next for each input. This produces an s_0-encoded output, so the resulting arrival times are singletons. For example, if y is the s_1-encoded input, the longest path through the cell starts at x (time 14), passes through f (time 16), and goes through the output mux (time 17; we assume that muxes have unit delay). This gives the (17) entry in the *Shannon* column in the y row.

Next, the algorithm considers a *Start* cell: one that starts a Shannon decomposition at each of the three inputs. For example, if we start a Shannon decomposition with s_0 input x, the two s_0 inputs for y and z are passed to copies of f (this is the structure of a *Start* cell), and the s_0 x input is passed to the three-wire output (*Start* cells produce an s_1-encoded output). The outputs of the two copies of f become the x_0 and x_1 outputs of the cell and arrive at time $8 + 2 = 10$ because z arrives at time 8 and f has a delay of 2. The s_0 x input is passed directly to the output, which we assume is instantaneous, so it arrives at the output at time 14. Together, these produce the arrival-time tuple (10, 10, 14), which is the entry in the x row in the *Start* column.

The *Stop* and *Extend* cases require one of their inputs to be s_1 encoded. Since no such encoding for y is available (i.e., there is no triplet in its arrival time set), we do not consider using y as this input; hence, the *Stop* and *Extend* columns are empty in the y row.

In Fig. 3, the *AT* function (called in lines 22–24) is used to compute the arrival time for each of these cases. In general, the arrival time $AT(\langle s_i, s_f, s_o\rangle, t, T, d(n))$ of a variant of node n with shape $\langle s_i, s_f, s_o\rangle$ depends on the arrival time t of the encoded input, the set of arrival times T of the other inputs, and the delay $d(n)$ of the combinational logic. It is computed using regular static timing analysis, i.e., (1) for each of the components of the cell. We do not present pseudocode for the *AT* function since it is straightforward yet fussy.

Even this simple example produced more than 13 arrival-time tuples from five (Fig. 9 does not list the higher order encodings our algorithm also considers); such an increase is typical. Fortunately, most are not interesting as they obviously lead to slower circuits. Since in this phase we are only interested in the fastest circuit, we can discard most of the results of this exhaustive analysis to greatly speed up the analysis of later nodes—we discuss this next.

G. Pruning FATs

As shown in Fig. 9, a node can usually be resynthesized in many ways. Fortunately, most variants are not interesting because they are slower than others. In this section, we describe our policy for discarding implementations that are never faster

$$\begin{tikzcd}[column sep=small]
(10,10,14) & & & & & \\
& & (15,15,11) & & & \\
(15) \arrow[r, phantom, "\preceq"] & (16) \arrow[ur, phantom, "\preceq"] \arrow[r, phantom, "\preceq"] & (17) \arrow[dr, phantom, "\preceq"] & & & \\
& & & (16,16,6) \arrow[r, phantom, "\preceq"] & (16,16,7) \arrow[r, phantom, "\preceq"] & (16,16,8)
\end{tikzcd}$$

Pruned FAT set: $\{(15), (10,10,14)\}$

Fig. 10. Pruning the FATs from Fig. 9.

than others since in the first phase we are only interested in whether there is a circuit that will run with period c or faster. In the second phase, we will use slower cells off the critical path to save area (see Section III).

If p and q are two arrival times at the output of a node, then we write $p \preceq q$ if an implementation of the circuit where the arrival time of the node is q cannot be faster (i.e., admit a smaller clock period) than an implementation where the arrival time of the node is p. Consequently, if we find cell implementations that produce p and q, we can safely ignore the implementation that produces q without fear of erroneously concluding that a particular period is unattainable. Our FAT set pruning operation removes all such dominated arrival times from the set of arrival times produced as described previously. In Fig. 3, this pruning is performed on lines 25 and 26.

In fact, we implement a conservative version of the $\preceq$ relation described previously because the precise condition is actually a global property. If a node is off the critical path, for example, it is probably the case that more arrival times could be pruned than our implementation admits, but practically we find that our pruning works quite well in practice.

For s_0-encoded signals, the ordering is simple: The faster-arriving signal is superior.

For two arrival times for signals encoded in the same way, the ordering is piecewise: If every component is faster, then the arrival time is superior; otherwise, the two arrival times are incomparable because a later Shannon decomposition might be able to take advantage of the differential in ways we cannot predict locally.

For arrival times corresponding to different encodings, the argument is a little more subtle. Consider Fig. 8. In general, only the first two wires in an encoded signal are ever fed directly to functional blocks (e.g., x_0 and x_1 in the s_3 encoding in Fig. 8 and the others are fed to a collection of multiplexers. The wires in higher level encodings must eventually meet more multiplexers than those in lower level encodings, so a lower level encoding whose elements are strictly better than the first elements in a higher level encoding is always going to be better.

Concisely, our choice of $\preceq$ (our pruning rule) is, for two arrival times $p = (p_0, p_1, \ldots, p_n)$ and $q = (q_0, q_1, \ldots, q_m)$ for potentially different encodings,

$$p \preceq q \qquad \text{iff } n \le m,\ p_0 \le q_0,\ p_1 \le q_1, \ldots, \text{ and } p_n \le q_n.$$

Fig. 10 illustrates how pruning reduces the size of the FAT set computed in Fig. 9. The singleton (15) dominates most of the other arrival times (some of which appear more than once in Fig. 10—remember that we ultimately operate on FAT sets, not on the table of Fig. 9), but (10, 10, 14) is not comparable with (15) (for a singleton to dominate a triplet, the first value in the triplet must be greater or equal).

That the eleven arrival times computed in Fig. 9 (only eight are distinct) boil down to only two interesting ones (Fig. 10) is typical. Across all the circuits we have analyzed, we find that pruned FAT sets seldom contain more than four elements. This is a key reason our algorithm is efficient: Although it is considering many circuit structures at once, it only focuses on the fastest ones.

H. Retiming

At this point, we have described our technique for restructuring circuits based on Shannon decomposition: We consider reimplementing isolated nodes with variants taken from a virtual library (Fig. 5 shows a subset) and discussed how we can represent these variants as FAT sets. We want now to choose variants so as to improve the effects of a later retiming step.

Retiming [1] 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 (noncritical) ones. However, it cannot decrease the total delay along a cycle.

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

$$\mathrm{lb}(S) = \max\left(\max_{\mathcal{C}\in\mathrm{cycles}(S)} \frac{d_{\mathcal{C}}}{r_{\mathcal{C}}}, \max_{\mathcal{P}\in\mathrm{paths}(S,\mathrm{spi},\mathrm{spo})} \frac{d_{\mathcal{P}}}{r_{\mathcal{P}}+1}\right) \quad (2)$$

is known as the fundamental limit of retiming.

Classical retiming may not achieve $\mathrm{lb}(S)$. To achieve it in general, we must allow registers to be inserted at precise points inside the nodes. We will assume this is possible (which it is, for example, in field-programmable gate arrays (FPGAs) [12]), so $\mathrm{ret}(S) = \mathrm{lb}(S)$ holds. We shall thus focus on transforming S to minimize $\mathrm{lb}(S)$.

I. Using Bellman–Ford to Compute Arrival Times

Computing $\mathrm{lb}(S)$ by enumerating cycles and applying (2) is not practical because the number of cycles may be exponential; instead we use the Bellman–Ford single-source shortest path algorithm,[1] where our source node is spi.

To apply Bellman–Ford, which has no notion of registers, we treat registers as nodes with negative delay: $\forall r \in R, d(r) = -c$, where c is the desired period. Only registers are assigned these artificial delays; the other nodes, i.e., the ones containing normal combinational logic, keep their positive delays $d(n)$ as defined before in Section II-A. Pan [2] also uses this technique.

[1]Bellman–Ford reports if a graph has any negative cycles or not. Only if all cycles are positive can it compute the shortest paths; it runs in $O(VE)$. Technically, we change signs, so we detect positive cycles instead of negative ones and compute the longest path instead of the shortest if all cycles are negative. Note that this is not solving the longest simple-path problem, which allows positive cycles and is known to be NP-complete.

Now, the total length of path $\mathcal{P} \in \mathrm{paths}(S)$ becomes $\sum_{n\in\mathcal{P}} d(n) = \sum_{n\in\mathcal{P}\setminus R} d(n) + \sum_{n\in\mathcal{P}\cap R} d(n) = d_{\mathcal{P}} - c \cdot r_{\mathcal{P}}$, where the first term is the delay of the combinational logic and the second corresponds to the registers.

For any path $\mathcal{P} \in \mathrm{paths}(S, \mathrm{spi}, \mathrm{spo})$, we have

$$c \geq d_{\mathcal{P}}/(r_{\mathcal{P}}+1) \Leftrightarrow d_{\mathcal{P}} - c \cdot r_{\mathcal{P}} \leq c \Leftrightarrow \sum_{n\in\mathcal{P}} d(n) \leq c.$$

Moreover, any cycle $\mathcal{C} \in \mathrm{cycles}(S)$ is a closed path, so

$$c \geq d_{\mathcal{C}}/r_{\mathcal{C}} \Leftrightarrow d_{\mathcal{C}} - c \cdot r_{\mathcal{C}} \leq 0 \Leftrightarrow \sum_{n\in\mathcal{C}} d(n) \leq 0.$$

Equation (2) becomes

$$c \geq \mathrm{lb}(S) \Leftrightarrow \begin{cases} \forall \mathcal{C} \in \mathrm{cycles}(S), & \sum_{n\in\mathcal{C}} d(n) \leq 0 \\ \forall \mathcal{P} \in \mathrm{paths}(S,\mathrm{spi},\mathrm{spo}), & \sum_{n\in\mathcal{P}} d(n) \leq c. \end{cases}$$

That is, the period $c \geq \mathrm{lb}(S)$ iff no cycle is positive, and the longest path from spi to spo is at most c. The first condition is verified if the Bellman–Ford algorithm converges to a bounded number of iterations. If so, it also gives us $\mathrm{at}(n)$—the longest path between spi and any node n. We verify the second condition by checking if $\mathrm{at}(\mathrm{spo}) \leq c$.

Therefore, $\mathrm{lb}(S)$ can be approximated by binary search on the period c.

To consider the combined effect of our restructuring and retiming, we use a variant of the Bellman–Ford algorithm that uses the FAT computation plus pruning operation as its central relaxation step (Fig. 3). The main loop (lines 7–13 in Fig. 3) terminates when the relaxation has converged to a solution or it has become fairly obvious that no solution will be found. This latter case is actually a heuristic, which we will discuss in the next section.

If Bellman–Ford converges, i.e., reaches a fixed-point such that $\mathrm{at}(\mathrm{spo}) \leq c$, then there exists an equivalent circuit for which $\mathrm{lb}(S) \leq c$, so, after retiming, c is feasible. To prove this claim, we simply build a circuit using the cell variants we considered during the relaxation procedure. However, such a brute-force construction produces overly large circuits, so instead we use a more clever construction that limits Shannon-induced duplication to critical paths only, which is the subject of Section III.

We illustrate our brute-force construction on the sample in Fig. 12. Convergence of our augmented Bellman–Ford algorithm implies a fixed-point solution, i.e., a FAT set for each node, that is stable under the pruned FAT set computation. For the sample in Fig. 12, Bellman–Ford converges to the fixed-point solution at the bottom of that figure, so we claim that 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 Fig. 5 and use any FAT elements at each input, as described in Section II-F.

For example, for node h at the bottom of Fig. 12, we consider two implementations. These are *Start* and *Shannon* (Fig. 5), both with g's output as the select. These give arrival times of (4, 4, 8) and (9).

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

J. Termination of Bellman–Ford and max-iterations

Our modified Bellman–Ford algorithm has three termination conditions: two that are exact and one that is a heuristic. The most straightforward is the check for a fixed point on line 12. This is the only case in which we conclude that the relaxation has converged and is usually reached quickly in practice.

The second termination condition is due to the topology of our circuit graphs. If no fixed point exists, the s_0-encoded arrival time in $\mathrm{fat}(\mathrm{spo})$ will eventually become greater than c. This is because any positive cycle (the absence of a fixed point means such a cycle exists) must pass through spo and along any positive cycle, the arrival times must keep increasing during the relaxation procedure. So, checking the s_0-encoded signal (there is always exactly one) suffices. This is the check in line 10.

The third termination condition—the hard iteration bound of *max-iterations* on line 7—is a heuristic. We employ such a bound because convergence usually happens quickly, while nonconvergence can be very slow. It is always safe to terminate the loop earlier, i.e., assume that the rest of the iterations, if continued, would have never converged: There is no inconsistency risk, but we may get a suboptimal solution.

We expect convergence to be fast because of the behavior of the usual Bellman–Ford algorithm. When it converges, it does so in at most n iterations, where n is the number of vertices in the graph.[2] Indeed, the smallest positive cycle in the graph has a length of at most n. However, provided that the vertices with positive weights are visited in a topological order (w.r.t. the dag obtained from the graph by removing vertices with negative weights), the convergence is in practice much faster. In our adapted Bellman–Ford algorithm, we use such an ordering for the inner loop at lines 8 and 9. Therefore, we expect fast convergence when period c is feasible. In particular, the speed of convergence, while depending on the circuit topology, should be essentially independent of the distance between the feasible clock period c tried and the lowest clock period achievable by our technique. However, when c is unfeasible and gets closer to the lowest achievable period, the number of iterations before overflow may increase arbitrarily.

The graphs in Fig. 11 provide empirical evidence for this. There, we plot iteration counts for three large circuit samples (with *max-iterations* arbitrarily set to 2000; we observe similar behavior across all circuits). The scale is logarithmic. For the first two examples, we do observe that divergence is indeed costly to detect when c gets close to the limit. In the third case, probably because of a tight cycle close to one circuit output, we have no such curve. However, what matters is that we observe that convergence is always very fast. Therefore, it makes sense to choose *max-iterations* to be a small quantity. We choose $max\text{-}iterations = 200$ and get good results for all our benchmarks, that is to say the algorithm never fails to obtain stable FAT sets because of an early timeout. As a result, if the minimum feasible c is computed by binary search, the result is very close to the true value.

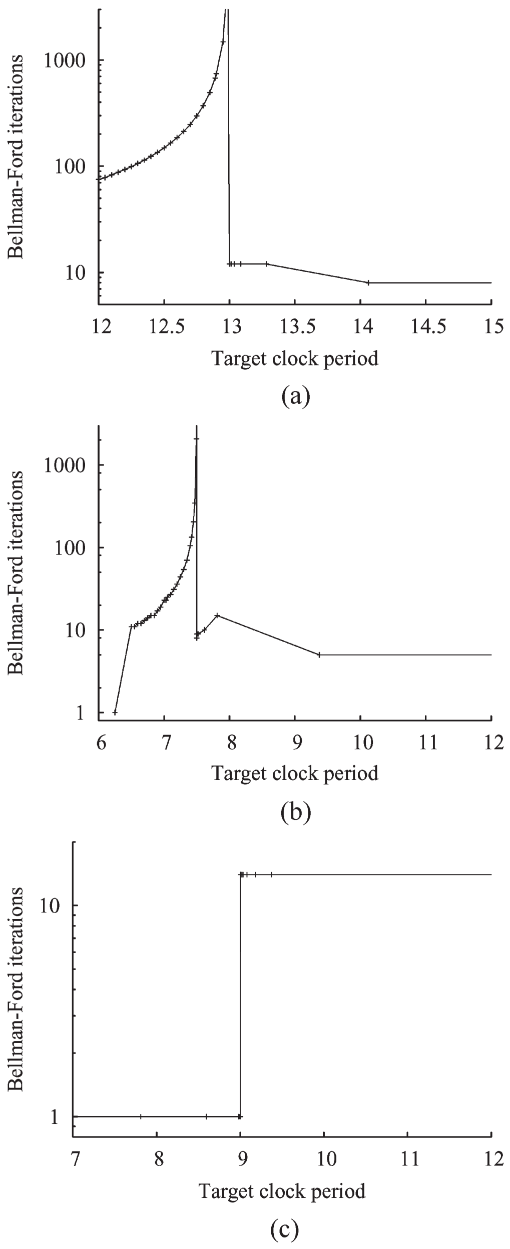


Fig. 11. Iterations near the lowest c. (a) s38417 circuit sample. (b) s9234 circuit sample. (c) s13207 circuit sample.

III. Second Phase: Resynthesizing the Circuit

The first phase of the algorithm focuses exclusively on performance: It considers only the fastest possible circuit it can find and ignores the others for efficiency. This makes sense since it is trying to establish the existence of a fast-enough circuit, but we would really like to find a fast-enough circuit that is as small as possible. The goal of the second phase of the algorithm, which is described in this section, is to select a

[2]Unfortunately, we do not have a similar result for our variant because of the complex behavior of FAT set generation and pruning. However, even if we knew a closed-form bound, we would still prefer to use our heuristic early termination condition because it produces very good results much faster.

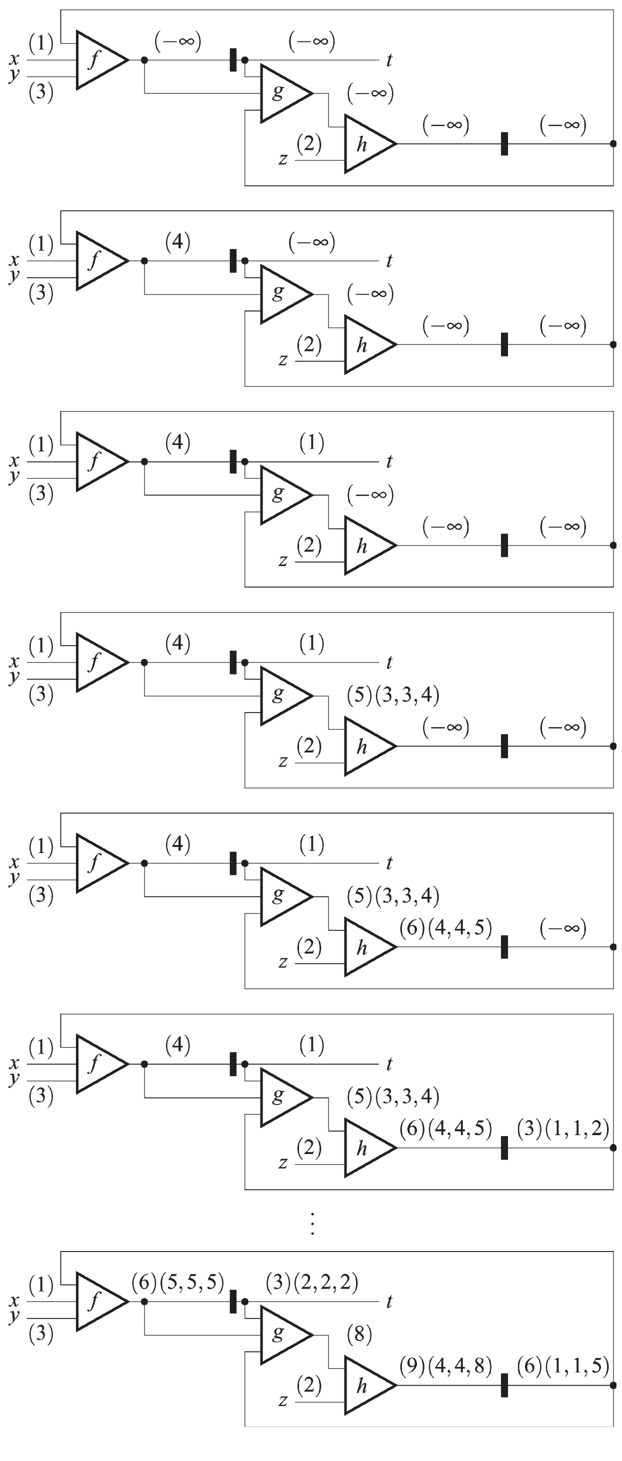


Fig. 12. Computing FATs for a circuit with desired period $c = 3$. Each gate has delay 2. The topmost circuit is the initial condition; the bottom is the fixed point. A total of seven complete passes over the circuit was required to find the fixed point; only the relaxation steps taken during the first are shown.

minimal-area circuit that still meets the given timing constraint. We do this heuristically.

The circuit produced by this part of our algorithm is often worse than the original. The circuit in Fig. 13(a) happens to have the same minimum period (9) as the original circuit in Fig. 12. However, $\mathrm{lb}(S)$ as defined by (2) is 3; thus, after retiming [Fig. 13(b)], the minimum period drops to our target of 3.

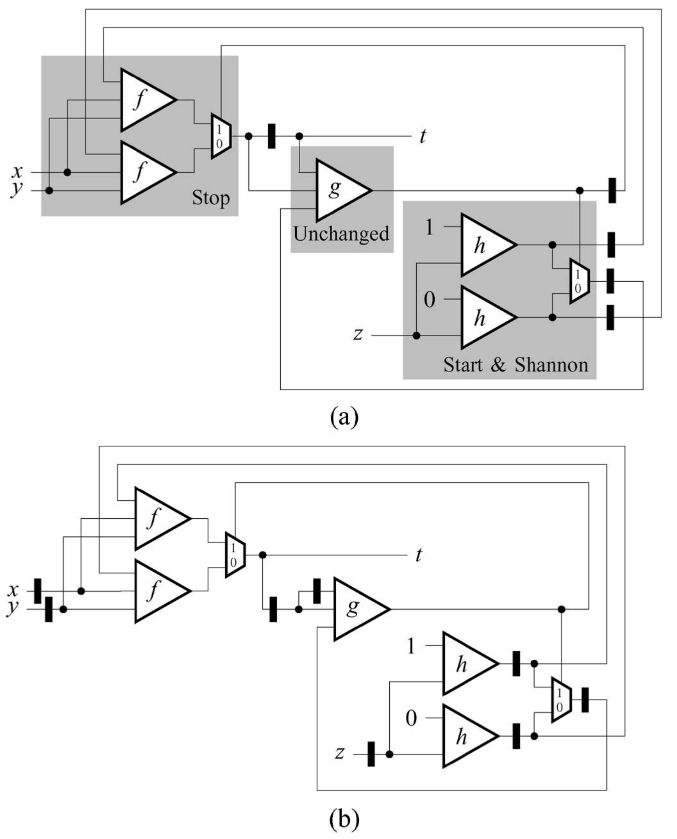


Fig. 13. (a) Circuit from Fig. 12 restructured according to the FAT sets. Its period (9) happens to be the same as that of the original; in general, it is different (and not necessarily better). (b) After retiming, the circuit runs with period 3 as desired.

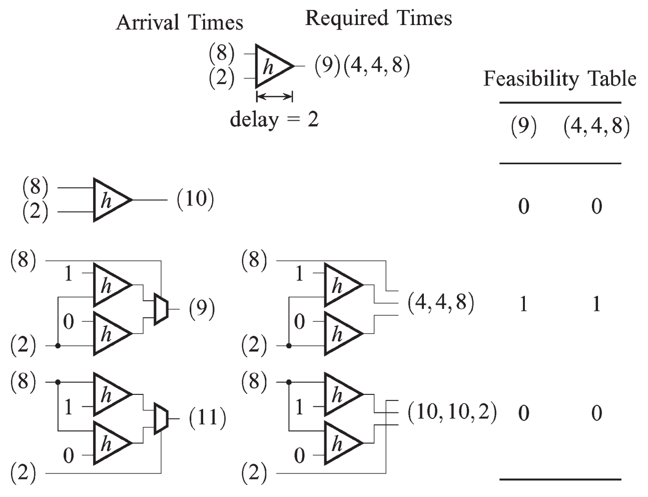


Fig. 14. Selecting cells for node h in Fig. 12.

To minimize area under a timing constraint—the main goal of the second phase—we use a well-known scheme from technology mapping: Nodes that are not along critical paths are implemented using smaller slower cells, in such a way that the overall minimum clock period is the same.

Our construction rebuilds the circuit starting at the primary outputs and register inputs and working backward to the primary inputs and register outputs. We insist that the circuit is free of combinational loops, so this amounts to a reverse topological order.

For each gate, we consider the FATs of its fan-ins computed in phase one and the feasible required times (FRTs) of its fan-outs, which we compute as part of the reconstruction procedure. Fig. 14 illustrates this for the h node from Fig. 12. The FRTs for primary outputs and registers are the FATs for these nodes computed in the first phase. For each gate, we construct a cell (occasionally more than one) that is just fast enough to meet the deadline (i.e., compute the outputs to meet the required times given the arrival times) without being larger than necessary.

An FRT set for a node is much like a FAT set: It consists of tuples of numbers that describe when each wire in an encoded signal is needed. At each node, we consider different cells for the node that produce the desired encoding (i.e., the arity of the FRT tuple). Since the FAT sets were produced by considering all possible cells at each node, we know that some cell will achieve the desired arrival time. To save area, we select the smallest such cell.

If we are lucky, an *Unchanged* cell suffices, meaning the functional block does not need to be duplicated and there are no additional multiplexers. In Fig. 13(a), node g appears as *Unchanged*.

For node f, the *Stop* cell was selected. Note that the signal from h to f uses an s_1 encoding.

For h, we actually selected two cells: *Start* and *Shannon*. Fortunately, they share logic—the duplicated h logic block. The s_1-encoded output of *Start* goes to f; the s_0 output of *Shannon* is connected to g.

Note that when a register appears on an encoded signal, it is simply replicated on each of its wires. In Fig. 13(a), for example, the register on the output of h became four: one on the output of the mux, an s_0-encoded signal, and three for the s_1-encoded signal, which are the two outputs from the two h blocks and the selector signal (the output of g). While this may seem excessive, the subsequent retiming step may remove many of these registers.

A. Cell Families

That we need multiple cells for a particular node to meet timing happens often enough that we developed a heuristic to reduce the area in such a case. It follows from a simple observation: Many of the cells in Fig. 5 are similar. In fact, some are subsets of others; so, if we happen to be able to use a cell and its subset to implement a particular node, it requires less area than if we use two different cells.

We call cells that only differ by the addition or deletion of multiplexers on the output members of a *family*. By this definition, each cell is a member of exactly one family. In Fig. 5, there are three such families: *Unchanged* is a family by itself, *Shannon* and *Start* is another family (they both encode one of their inputs), and *Extend* and *Stop* are the third (each takes a single s_1-encoded input). Families with cells with higher order encodings are larger.

To save area, we try to use only cells taken from the same family since each cell in a family can also be used to implement others in the family without making additional duplicates of the node's function.

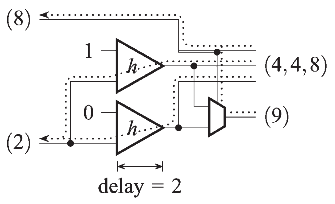


Fig. 15. Propagating required times from outputs to inputs. Only the critical paths (dotted lines) impose constraints.

B. Cell Selection

Working backward from the outputs and register inputs, we repeat the enumeration step from the first phase (e.g., Fig. 9) at each node to generate the set of all cells that are functionally correct for the node (i.e., have the appropriate input and output encodings). From this set, we try to select a set of cells that both are small and can meet the timing constraint at the node. Fig. 14 illustrates this process for the h node in Fig. 12.

We consider the cells generated by the enumeration step one family at a time in order of increasing area. Practically, we build a feasibility table whose rows represent cell families and whose columns represent required times. Such a table appears on the right side of Fig. 14. Our goal with this table is to select the minimum number of rows with small area to cover all the required times for the node. We consider rows instead of cells because implementing multiple cells from the same family is only as costly as implementing the largest cell in the family.

An entry in the table is 1 if some cell in the row's family satisfies the required time of the column. To evaluate this, we construct each possible cell for the node (such as those on the left side of Fig. 14) and calculate the arrival times of the outputs from the arrival times of the inputs. An arrival time satisfies a required time if it corresponds to the same encoding and the arrival time components are less than or equal to the required times. Note that these criteria are simpler than the $\preceq$ relationship used in the pruning.

We select several rows of minimum area that cover all columns (i.e., a collection of cell families that contain cells that are fast enough to meet the timing constraint). As mentioned, there is usually a row that covers all columns, so we usually pick that. Otherwise, we continue to add rows until all columns are covered. Fig. 14 is typical: The second row covers all columns, so we select it.

Selecting a cell in this way is a greedy algorithm that does not consider any later ramifications of each choice. We have not investigated more clever heuristics for this problem, although we imagine that some developed for performance-oriented technology mapping would be applicable.

C. Propagating Required Times

Once we have selected a set of families that cover all of the required time constraints (i.e., solved the feasibility problem), we construct all cells in these families (sharing logic within each family) and connect them to the appropriate fan-outs. In Fig. 14, we choose the second row and build both cells in that family. Fig. 15 shows this: A single pair of h nodes are built,

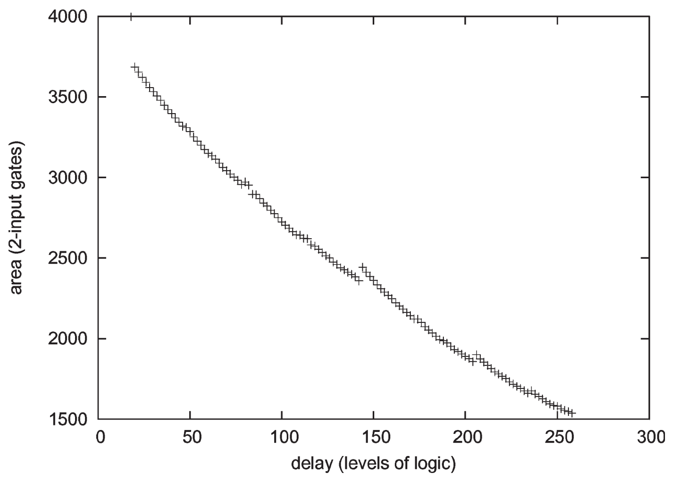


Fig. 16. Performance/area tradeoff obtained by our algorithm on a 128-bit ripple-carry adder.

but both an s_1-encoded signal is produced, that with required time (4, 4, 8), and an s_0 signal with required time (9).

At this point, we now have a circuit that includes cells implementing the node we are considering. Because we built them earlier, we know the required times at the inputs to each the fan-outs, so we work backward from these times to calculate the required times at the inputs to our newly created cells. This is simple static timing analysis on a real circuit fragment (i.e., we now only have wires and gates, not cleverly encoded signals).

Fig. 15 illustrates how we propagate required times at the outputs of a cell back to the inputs. In this case, it is easy because we do not need to consider encoded signals specially. This is standard static timing analysis. For example, the (4, 4, 8) constraint on one of the outputs means that the inputs to each copy of h must be available at time 2 (h has a delay of 2).

Note that in general, required times may differ from arrival times because of slack in the circuit. We do not see this in this example since it is small and h is on the critical path, but it does happen quite frequently in larger examples. Again, we take advantage of this to reduce area.

We may now have several required times for each input of a family of cells, but the encoding must be the same for a particular input because they are all in the same family. In this case, we simply compute the overall required time of each input by taking the pairwise minimum, and we place it on the corresponding fan-in.

If two or more cell families are built, several required times with different encodings will be placed on the fan-ins. In this case, the fan-in nodes will see the current node as two or more distinct fan-outs instead of one. Such complex situations rarely occur in practice.

IV. Experiment

We implemented our algorithm in C++ using the SIS libraries [13] to handle Berkeley Logic Interchange Format files. Our testing platform is a 2.5-GHz 512-MB Pentium 4 running on Fedora Core 3 Linux.

TABLE I
Experiments on ISCAS89 Sequential Benchmarks

	Reference		Retimed		Ours		Time	Speed	Area
	period	area	period	area	period	area	(s)	up	penalty
s382	7	103	7	103	5	115		40%	11%
s386	7	85	7	85	7	85			
s400	8	103	7	103	6	111		16%	7%
s420	8	70	8	70	7	71		14%	1%
s444	8	101	7	101	6	106		16%	5%
s510	8	184	8	184	8	184	0.5		
s526	5	113	5	113	5	113			
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			
s838	10	154	10	154	8	162	2.6	25%	5%
s1196	9	365	9	365	9	365	0.6		
s1238	9	338	9	338	9	338			
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		
s5378†	8	819	7	819	7	819	1.4		
s9234	11	662	8	656	8	684	6.7		
s13207	14	1382	11	1356	9	1416	18.0	22%	4%
s15850†	20	2419	14	2413	10	2530	4.7	40%	5%
s35932†	4	3890	4	3890	3	4978	3.9	33%	28%
s38417	14	7706	14	7652	13	7871	113	7%	3%
s38584†	14	7254	13	7220	11	7296	56	18%	1%

† These circuits could not be processed by SIS's *full_simplify* algorithm—part of *script.rugged* —within 8 hours. We used *simplify* instead, therefore the starting points are poorer compared to the other samples.

A. Combinational Circuits

While our algorithm is most useful on sequential circuits, it can also be applied to purely combinational circuits. However, classical combinational performance optimization techniques, such as the *speedup* function in SIS, outperform our technique because they consider more possible transforms than combinations of Shannon decompositions. In particular, the best ones consider the functions of the nodes and perform Boolean manipulations. Our algorithm treats functional nodes as black boxes, which both greatly reduces the space of optimizations we consider and greatly speed ups our algorithm.

Our algorithm does perform well on certain combinational circuits. Fig. 16 shows how it is able to trade area for reduced delay with a 128-bit ripple-carry adder. For this example, we varied the requested c and generated a wide spectrum of adders, ranging from the original ripple-carry at the lower right to a full carry-select adder at the upper left. Our algorithm makes evaluating such a tradeoff practical: it only took 22 s to generate all 122 variants.

B. Sequential Benchmarks

We ran our algorithm on mid- and large-sized ISCAS89 sequential benchmarks and targeted an FPGA-like, three-input lookup-table architecture. Hence, we report delay as levels of logic and area as the number of lookup tables.

Following Saldanha *et al.* [7], 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* [14]. We report the results of this classical FPGA delay-oriented flow in the Reference column in Table I.

Starting from these optimized circuits, we compare running retiming alone (*retime -n -i*, which is modified to use the unit delay model) with running our algorithm followed by retiming. The Retimed and Ours columns list the period and area results. Our running time, which is listed in the Time column, includes finding the minimum period by binary search, so this actually includes multiple runs of our algorithm. We verified the sequential equivalence of the input and output of our algorithm using Verification Interacting with Synthesis [15]; our reported times do not include this.

Although our algorithm does nothing to half of the examples, it provides a significant speedup 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., 70 MB for the largest example s38417. Our technique therefore appears to scale well.

That our algorithm is not able to improve certain circuits is not surprising. Our algorithm is fairly specialized (compared to, say, retiming, and only attacks time-critical feedback loops that have small cuts (the loops can be broken by cutting only a few wires). Circuits on which we show no improvement may have wide feedback loops (e.g., the programmable-logic-array-style next-state logic of a minimum-length encoded finite-state machine or a multibit "arithmetic" feedback) or may be completely dominated by feedforward logic (e.g., simple pipelined data paths).

V. Conclusion

We presented an algorithm that systematically explores combinations of Shannon decompositions while taking into account a later retiming step. The result is a procedure for resynthesizing and retiming a circuit under a timing constraint that can produce faster circuits than Shannon decomposition and retiming run in isolation. 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.

Our algorithm runs in three phases: It first attempts to find a collection of FATs that suggests that a circuit exists with the requested clock period. If successful, it then resynthesizes the circuit according to these arrival times and heuristically limits duplication of nodes off the critical path to reduce the area penalty. Finally, the resynthesized circuit is retimed to produce a circuit that meets the initial timing constraint (a minimum clock period).

Experimental results show that our algorithm can significantly improve the speed of certain circuits with only a slight increase in area. Its running times are small, suggesting that it can scale well to large circuits.

References

[1] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.
[2] P. Pan, "Performance-driven integration of retiming and resynthesis," in *Proc. DAC*, 1999, pp. 243–246.
[3] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, "Logic decomposition during technology mapping," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 16, no. 8, pp. 813–834, Aug. 1997.
[4] K. J. Singh, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Proc. ICCAD*, 1988, pp. 282–285.
[5] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. Trevillyan, "Efficient techniques for timing correction," in *Proc. ISCAS*, 1990, pp. 415–419.
[6] P. C. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. K. Sahni, "Performance enhancement through the generalized bypass transform," in *Proc. ICCAD*, 1991, pp. 184–187.
[7] A. Saldanha, H. Harkness, P. C. McGeer, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Performance optimization using exact sensitization," in *Proc. DAC*, 1994, pp. 425–429.
[8] K. J. Singh, "Performance optimization of digital circuits," Ph.D. dissertation, Univ. California, Berkeley, CA, 1992.
[9] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 10, no. 1, pp. 74–84, Jan. 1991.
[10] S. Hassoun and C. Ebeling, "Architectural retiming: Pipelining latency-constrained circuits," in *Proc. DAC*, 1996, pp. 708–713.
[11] M.-C. V. Marinescu and M. Rinard, "High-level automatic pipelining for sequential circuits," in *Proc. ISSS*, 2001, pp. 215–220.
[12] H. Touati, N. Shenoy, and A. L. Sangiovanni-Vincentelli, "Retiming for table-lookup field-programmable gate arrays," in *Proc. Int. Workshop FPGAs*, 1992, pp. 89–93.
[13] E. M. Sentovich *et al.*, "SIS: A system for sequential circuit synthesis," Univ. California, Berkeley, CA, Tech. Rep. UCB/ERL M92/41, 1992.
[14] H. Touati, H. Savoj, and R. K. Brayton, "Delay optimization of combinational logic circuits by clustering and partial collapsing," in *Proc. ICCAD*, 1991, pp. 188–191.
[15] R. K. Brayton *et al.*, "VIS: A system for verification and synthesis," in *Proc. Comput.-Aided Verif.*, 1996, pp. 428–432.

Cristian Soviani received the degree from Bucharest Polytechnic Institute, Bucharest, Romania, in 1999. He is currently working toward the Ph.D. degree in the Department of Computer Science, Columbia University, New York, NY.

His research interests include sequential logic synthesis and optimization, high-level synthesis for high-performance network devices, embedded system design, and FPGAs.

Olivier Tardieu received the degrees from the École Polytechnique, Paris, France, in 1998, and the École des Mines, Paris, in 2001, and the Ph.D. degree in computer science from the Institut National de Recherche en Informatique et en Automatique, Sophia-Antipolis, France, and the École des Mines in 2004.

In 2005, he joined the Department of Computer Science, Columbia University, New York, NY, where he is currently a Postdoctoral Research Scientist. His research interests include programming language design, compilers, software safety, concurrency theory, and hardware synthesis.

Stephen A. Edwards (S'93–M'97–SM'06) received the B.S. degree from California Institute of Technology, Pasadena, in 1992, and the M.S. and Ph.D. degrees from the University of California, Berkeley, in 1994 and 1997, respectively, all in electrical engineering.

After a three-year stint with Synopsys, Inc., Mountain View, CA, in 2001, he joined the Department of Computer Science, Columbia University, New York, NY, where he is currently an Associate Professor. His research interests include embedded system design, domain-specific languages, and compilers.