Telescopic Units: A New Paradigm for Performance Optimization of VLSI Designs

Luca Benini, Enrico Macii, *Member, IEEE*, Massimo Poncino, *Member, IEEE*, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—This paper introduces a novel optimization paradigm for increasing the throughput of digital systems. The basic idea consists of transforming fixed-latency units into variable-latency ones that run with a faster clock cycle. The transformation is fully automatic and can be used in conjunction with traditional design techniques to improve the overall performance of speed-critical units. In addition, we introduce procedures for reducing the area overhead of the modified units, and we formulate an algorithm for automatically restructuring the controllers of the data paths in which variable-latency units have been introduced. Results, obtained on a large set of benchmark circuits, show an average throughput improvement exceeding 27%, at the price of a modest area increase (less than 8% on average).**

***Index Terms*— Circuit optimization, design automation, high-speed integrated circuits, logic design, synchronization.**

I. Introduction

THE ever increasing clock frequency of high-performance systems pushes IC designers and synthesis tools to perform substantial efforts in minimizing the delay of combinational logic blocks that constrain the cycle time. Gate-level timing optimization is often a computationally intensive task and sometimes it leads to a significant area and power-consumption overhead. In addition, it may not be the most convenient choice if some flexibility is allowed in changing the design architecture.

In the majority of circuit and system designs, *throughput* provides a more meaningful measure of performance than clock frequency. Throughput is abstractly defined as the amount of computation performed in a time unit. Obviously, decreasing the clock cycle time is one way to improve the throughput in a digital system. However, architectural optimizations such as parallelism exploitation and pipelining are much more effective in increasing throughput than bare clock speedup [1].

A well-known throughput-enhancement technique is based on using variable-latency units [1], [2]. For example, high-performance hardware components for division or for the computation of transcendental functions are well suited for variable-latency implementation. Such units complete execution in a variable number of clock cycles, depending on the input data they receive. The variable-latency implementation is a natural solution for floating-point arithmetic computations because the algorithms involved are iterative in nature and the number of iterations is data-dependent.

The basic principle that motivates the implementation of a variable-latency resource is that of "speeding up the common case" [1]. A fixed-latency unit completes execution with the latency of its longest possible computation. On the contrary, a variable-latency unit adapts its latency to the length of the computation it is performing. Average throughput is improved if the probability of a long-latency computation is much smaller than that of a short-latency one. Unfortunately, the overhead that occurs when instantiating a variable-latency unit is twofold. First, a *completion signal* must be provided to inform the environment of the termination of a computation. Second, the control logic in the environment must be able to synchronize with a variable-latency completion. Clearly, the overhead should be kept as small as possible.

High probability of short-latency computation and low overhead are the two conflicting requirements for the success of a variable-latency resource in satisfying the design goals. Hence, hand-crafted design of variable-latency units is a difficult task and computer-aided design tools may be of great help.

In this work, we address the automatic synthesis of high-throughput, variable-latency units and the estimation of the expected performance improvements. We focus our attention on components of synchronous circuits, which originally implement arbitrary combinational functions in a single clock cycle (i.e., with fixed latency of one unit). We transform such units into variable-cycle implementations, which we call *telescopic units*. Whereas such units have data-dependent latency, their clock rate can be sped-up to match the "common case," i.e., the critical-path delay of most computations that can still be achieved in one clock cycle. Longer computations will be split over two (or more) cycles. The overall performance improvement of this transformation stems from achieving a faster clock rate for the synchronous circuit of interest.

Seen as a black box, a telescopic unit produces two outputs: the functional output (i.e., the result of the computation) and a handshaking *hold signal* which is activated when the functional unit requires more than one clock cycle to complete the computation. The advantage of the telescopic unit is an increase in average throughput. The overhead consists of the

Manuscript received August 28, 1997. This work was supported in part by NSF under Contract MIP-9313701 and by ARPA under Grant DABT63-95-C-0049. This paper was recommended by Associate Editor F. Brglez.

L. Benini and G. De Micheli are with Stanford University, Computer Systems Laboratory, Stanford, CA 94305 USA.

E. Macii and M. Poncino are with Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy 10129.

Publisher Item Identifier S 0278-0070(98)04630-2.

circuitry for the generation of the hold signal. Additional circuitry is required in the external control logic in order to observe the hold signal and synchronize the telescopic unit with its environment.

Although telescopic units are, in principle, similar to self-timed units [3], they operate in a fully synchronous environment. Hence, they take an integer number of clock cycles to complete their executions. The fully synchronous operation allows us to ignore the issues related to hazards, which make the design of large scale self-timed circuits complex and expensive.

We outline algorithms and heuristics for automatically synthesizing telescopic units which rely on symbolic techniques for exact timing analysis [4]. Experimental results confirm the viability of our approach, and they clearly indicate the applicability of the technique for throughput optimization, as well as for area optimization under throughput constraints.

Recently, Hassoun and Ebeling have presented an approach similar to ours, called *architectural retiming* [5]. Their idea is to increase the number of registers on latency-constrained paths, thus decreasing the cycle time without increasing the latency. This is made possible by adding a *negative register* to each newly-added register, in such a way that regular/negative register pairs are implemented as wires. The implementation of the negative registers is key for the applicability of the method. Since the output of a negative register is equal to the input value at the next clock cycle, the implementation requires a sort of prediction. This prediction is verified one clock cycle after its calculation: if it is correct, the system can proceed with the next prediction; otherwise, the mispredicted value must be flushed and the circuit must be restored to the previous state. This implies a one-cycle latency penalty.

Another similar approach that finds application in the design of asynchronous data-path units is called *speculative completion* [6]. This method merges the advantages of other well-known techniques for the detection of the computation completion for asynchronous units. The idea is that of associating multiple, "speculative" delay models with a unit, in such a way that the completion of an operation is detected in parallel with the unit itself. The multiple models account for different (e.g., worst-case verses best-case) speeds of early completion, and each speculative delay has its own *abort detection* logic that signals whether the corresponding delay model has to be aborted. Both architectural retiming and speculative completion are hand-crafted techniques that have not been automated. On the contrary, telescopic units are automatically synthesized.

The remainder of this manuscript is organized as follows. Section II provides the notation and some background information concerning delays in combinational circuits and summarizes how exact timing analysis can be performed efficiently using symbolic techniques based on algebraic decision diagrams (ADD's). Section III introduces the telescopic unit architecture, and Section IV discusses in detail algorithms and heuristics for the automatic synthesis of telescopic units. Section V addresses the problem of designing controllers for systems containing telescopic units. Section VI reports the results of a large set of experiments we have carried out on standard benchmark examples. Finally, Section VII is devoted to conclusions.

II. Background

A. Circuits and Delays

A *combinational circuit* is a feedback-free network of combinational logic gates, called gates for brevity. If the output of a gate g_i is connected to an input of a gate g_j, then g_i is a *fanin* of g_j and gate g_j is a *fanout* of gate g_i. A *controlling value* at a gate input is the value that determines the value at the output of the gate independent of the other inputs, while a *noncontrolling value* at a gate input is the value whose presence is not sufficient to determine the value at the output of the gate. For example, zero is the controlling value for a NAND gate.

Each connection c is associated with two delays, $d_r(c)$ rise delay and $d_f(c)$ fall delay. The *delay function* of connection c from gate h to gate g is called $d(c,x)$. It equals $d_r(c)$ if g takes value 1 when input vector x is applied to the primary inputs of the circuit. Otherwise, $d(c,x) = d_f(c)$. If all fanin connections of g have the same values of $d_r(c)$ and $d_f(c)$, we define the delay function of g as $d(g,x) = d(c,x)$, where c is any fanin connection of g. If $f(g,x)$ is the global function of g (function in terms of the primary inputs) and c connects gate h to gate g, then

$$d(c,x) = f(g,x) \cdot d_r(c) + f'(g,x) \cdot d_f(c).$$

Given a gate g, the *arrival time* $AT(g,x)$ is the time at which the output of g settles to its final value if input vector x is applied at time zero.

A *path* in a combinational circuit is a sequence of gates and connections $(g_0, c_0, \cdots, c_{n-1}, g_n)$, where connection $c_i, i = 0, 1, \cdots, n-1$ connects the output of gate g_i to the input of gate g_{i+1}. The *length* of a path $P = (g_0, c_0, \cdots, c_{n-1}, g_n)$ is defined as $d(P,x) = \Sigma_{i=0}^{n-1} d(c_i, x)$. The *topological delay* of a combinational circuit is the length of its longest path. An *event* is a transition $0 \rightarrow 1$ or $1 \rightarrow 0$ at a gate. Given a sequence of events, $(e_0, e_1, \cdots, e_n)$ occurring at gates $(g_0, g_1, \cdots, g_n)$ along a path, such that e_i occurs as a result of event e_{i-1}, the event e_0 is said to *propagate* along the path. Under a specified delay model, a path $P = (g_0, c_0, \cdots, c_{n-1}, g_n)$ is said to be *sensitizable* if an event e_0 occurring at gate g_0 can propagate along P. A *false path* is a nonsensitizable path. The *critical path* of a combinational circuit is the longest sensitizable path under a specified delay model: its length is the delay D of the combinational circuit and it is a lower bound on the cycle time T, i.e., $D \leq T$. For the sake of simplicity, we neglect set-up and hold times and propagation delays through registers. These factors can be easily incorporated into our analysis and synthesis technique.

B. Pseudo-Boolean Functions and ADD's

In the remainder of this manuscript, we assume the reader is familiar with the fundamental concepts of Boolean functions. In addition, we take for granted the knowledge of symbolic

techniques for the representation and the manipulation of such functions through binary decision diagrams (BDD's) [7]. Therefore, in the following, we only briefly recall the basic notions related to pseudo-Boolean functions and to the data structures, the algebraic decision diagrams (ADD's) [8], which are commonly used for their representation.

A n-input pseudo-Boolean function f: $B^n \rightarrow S$ is a mapping from a n-dimensional Boolean space to a finite set S. Function f can be efficiently stored and manipulated through an ADD, an extension of the BDD which allows values from an arbitrary finite domain to be associated with the terminal nodes (i.e., the *leaves*) of the diagram.

Among the existing operators for efficient ADD manipulation, *THRESHOLD* is of particular importance for our purposes. It takes two arguments: f a generic ADD, and *val* a threshold value. It sets to 0 all the leaves of f whose value is smaller than *val* and to 1 all the leaves of f whose value is greater than or equal to *val*. The resulting ADD, f_{val} is thus restricted to have only 0 or 1 as terminal values; therefore, it is a BDD.

C. ADD-Based Timing Analysis

The problem of calculating the timing response of a combinational circuit can be formulated as follows [4]. Given the circuit, find the set of input vectors for which the length of the critical path, under a specified mode of operation and a gate delay model, is maximum. The length of the critical path gives the overall circuit delay.

Consider a gate g of the network and a primary input vector $x \in X$, where X is the set of all the *care* input vectors of the circuit. The arrival time at its output line $AT(g,x)$ is evaluated in terms of the arrival times of its inputs and the delays of its fanin connections $d(c_j,x)$. Let c_j be the connection to pin j of gate g.

If all fanins of g have noncontrolling values

$$AT(g,x) = \max_j \{AT(c_j,x) + d(c_j,x)\}.$$

If at least one fanin c_j of g has a controlling value for input $x \in X$, where X is the set of all possible care input vectors

$$AT(g,x) = \min_j \{AT(c_j,x) + d(c_j,x) | c_j = \text{controlling}\}.$$

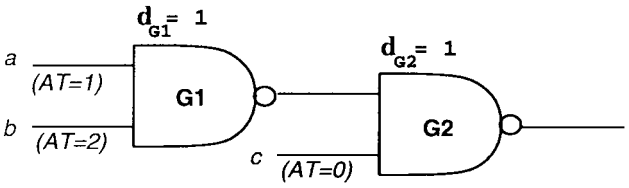
Finally, if $x \notin X$

$$AT(g,x) = -\infty.$$

Differently from what happens with traditional delay analyzers, the use of the ADD-based timing analysis tool has made it possible to compute and store the length of the critical path for *each* input vector.

The availability of the complete timing information regarding the combinational circuit is essential for the realization of the synthesis algorithms described in Section IV.

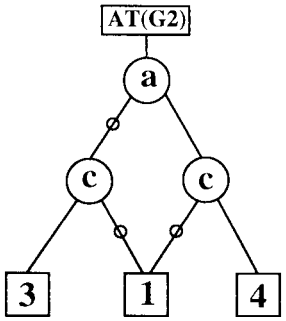
Example 1: Consider the combinational circuit of Fig. 1(a), and assume, for simplicity, the connection delays for a single gate g_i to be the same for all fanins and input vectors and to be given as single values d_{g_i} for each gate. By applying the three equations above on a gate-by-gate basis, and proceeding from the inputs to the outputs of the circuit, we can determine the arrival time of the output node of gate $G2$ for each input vector. The table of Fig. 1(b) provides such information. It is now possible to efficiently store the content of the table as an ADD. Fig. 1(c) depicts the final data structure.



(a)

a	b	c	AT(G2)
0	0	0	1
0	0	1	3
0	1	0	1
0	1	1	3
1	0	0	1
1	0	1	4
1	1	0	1
1	1	1	4

(b)



(c)

Fig. 1. (a) A combinational circuit, (b) its output arrival times, and (c) the corresponding ADD.

D. Throughput and Latency

The *throughput* P of a unit is defined as the amount of computation (i.e., the number of times a new output value is produced) carried out per time unit. For instance, the throughput of a combinational logic circuit with delay of 15 time units is $P = 1/15$. The *latency* L of a digital system is defined as the number of clock cycles required for a computation to complete. A fixed-latency unit with latency L clocked with period T has constant throughput $P = 1/(LT)$.

For variable-latency units, we consider the *average* latency L_{ave} over a period of time $T_{tot} \gg T$. The average throughput is simply $P_{\text{ave}} = 1/(L_{\text{ave}}T)$. In the following sections, we use the shorthand notation L and P as opposed to L_{ave} and P_{ave} to denote average latency and throughput, respectively.

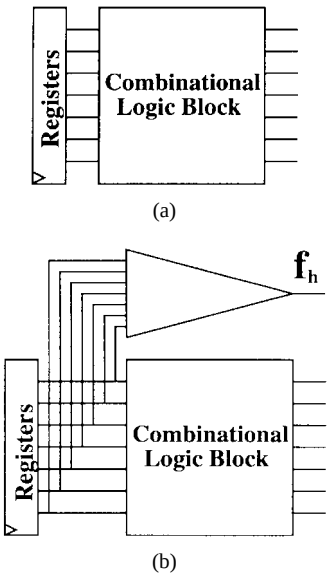


Fig. 2. (a) A combinational unit and (b) a telescopic unit.

III. Telescopic Unit Architecture

Consider the problem of increasing the throughput of a combinational unit, such as the one shown in Fig. 2(a). This can be done by shortening the cycle time of the unit from its original value T to $T^* < T$. One possible way of providing functional correctness is to extend the unit to provide an additional output signal f_h which is asserted for all input patterns requiring more than T^* time units to propagate to the outputs of the block [see Fig. 2(b)].

We call *telescopic unit* the modified unit, since it may require $L_{\max} > 1$ cycles to complete its execution, depending on the specific patterns appearing at its primary inputs. We consider here the situation in which $L_{\max} = 2$. In this case, the computation completes in T^* time units for patterns such that $f_h = 0$ and in $2T^*$ time units for patterns such that $f_h = 1$.

The average throughput of the original unit is

$$P = \frac{1}{T}. \tag{1}$$

For the telescopic unit, the lower the probability of the *hold* signal f_h to take on the value 1, the larger the overall throughput improvement. In fact, its average throughput P^* is given by

$$P^* = \frac{\mathrm{Prob}(f_h)}{2T^*} + \frac{1 - \mathrm{Prob}(f_h)}{T^*} \tag{2}$$

where $\mathrm{Prob}(f_h)$ is the probability of the hold signal to be one. Thus, the use of the telescopic unit is advantageous only for some values of T^* and $\mathrm{Prob}(f_h)$, i.e., when $P^* > P$. More specifically, we can write the following condition for throughput improvement

$$\mathrm{Prob}(f_h) < \frac{2(T - T^*)}{T}. \tag{3}$$

Clearly, Inequality 3 is valid only for $T^* \geq T/2$ since we have made the assumption that $L_{\max} = 2$[1]. In order to automatically synthesize telescopic units, two problems must be solved. First, we need to compute and synthesize the hold function, a combinational logic function that detects *all* input patterns that propagate to the outputs with delay larger than T^*. Second, we must modify the controller of the datapath where the telescopic unit is instantiated. The modified controller synchronizes the environment with the telescopic unit by delaying subsequent computations when $f_h = 1$. The following two sections deal with these problems in detail.

IV. Synthesis Algorithms and Heuristics

The computation of the arrival time ADD for a combinational unit allows us to determine all input vectors for which the propagation through the unit will be slower than T^*. This information is exploited to synthesize the logic which generates the proper values of the hold output.

Given the arrival time ADD of output O_i, $AT(g_{O_i}, x)$, the BDD for the function $f_h^{O_i}$ which assumes the value 1 for all the input vectors for which the arrival time of O_i is greater than the desired cycle time T^* is computed as

$$f_h^{O_i}(x) = THRESHOLD(AT(g_{O_i}, x), T^*). \tag{4}$$

Since we are interested in the set of input conditions for which *at least* one output of the unit has an arrival time greater than T^*, we have that the *hold function* f_h can be easily determined as

$$f_h(x) = \bigvee_{i=1}^{m} THRESHOLD(AT(g_{O_i}, x), T^*) \tag{5}$$

where m is the total number of outputs.

Clearly, the key issue for making telescopic units usable in practice concerns the way f_h is implemented by the *hold circuit*. There are three main constraints that the final implementation of f_h must satisfy, and thus require particular consideration during synthesis. They are listed below in decreasing order of importance.

- The arrival time of output f_h must be strictly smaller than T^* for any possible input pattern. Otherwise, the telescopic unit cannot be guaranteed to work correctly.
- The probability of f_h to assume the value 1 must be small enough to guarantee an average throughput improvement, that is, $P^* > P$ (Inequality 3).
- The area and power of the hold circuit implementing f_h must be kept under control.

[1]The extension to $L_{\max} > 2$, not discussed in detail for the sake of clarity, is conceptually straightforward, but more complex to implement. This is because several hold signals $f_h^1, f_h^2, \cdots, f_h^k$ are required to make the unit work correctly. Function f_h^j takes on the value one for all the input patterns that require $(j+1)$ cycles to complete the execution. The expression for P^* can then be modified to account for values of $T^* < T/2$ as follows:

$$P^* = \frac{\mathrm{Prob}(f_h^k)}{(k+1)T^*} + \frac{\mathrm{Prob}(f_h^{k-1})}{kT^*} + \cdots \frac{\mathrm{Prob}(f_h^1)}{2T^*} + \frac{1 - \mathrm{Prob}(f_h^1 + f_h^2 + \cdots f_h^k)}{T^*}$$

where $\mathrm{Prob}(f_h^j)$ represents the probability that hold function $f_h^j = 1$.

It should be observed that the ON-set of f_h, as defined by (5), contains *all* and *only* those input patterns that propagate to the output with delay larger than T^*. Hence, any implementation of the hold function must *cover* the ON-set of f_h, but it may also include other input conditions. By enlarging the hold conditions, a faster and smaller hold circuit may be obtained. Functional correctness is preserved, but the average throughput of the telescopic unit (5) may be degraded, because the circuit will hold for some input patterns with propagation delay smaller than T^*. Obviously, if some input *don't care* conditions are known, they can be profitably exploited to enlarge f_h without affecting the throughput of the unit.

We have exploited this observation to formulate two heuristic algorithms, described in the next two sections, whose target is to determine an *enlarged hold function* $f_h^e \geq f_h$, such that P^* only marginally degrades, but the implementation of f_h^e meets the timing constraint T^* and has a limited area. They both start from the BDD representation of f_h. The first method generates the hold logic following an iterative paradigm. First, the BDD of f_h is mapped onto a multiplexor network. Then, such network is optimized through traditional logic synthesis techniques. Finally, a check is made to find out if the timing constraint $T(f_h) < T^*$ is met. If this is not the case, the ON-set of f_h is enlarged to obtain f_h^e by properly removing some BDD nodes, and the process is repeated. The second heuristics produces a *sum-of-products* (SOP) description of f_h^e directly from the BDD of the initial f_h. The first heuristic is fast and works well for many examples, while the second is more computationally intensive and should be used when high-quality results are desired (i.e., maximum throughput improvement and minimum area overhead). Both methods are described in detail in the following sections.

A. BDD-Based Heuristics

The starting point of this method is the BDD of f_h as defined by (5). We search for a new hold function $f_h^e \geq f_h$ whose implementation satisfies the timing constraint

$$THRESHOLD[AT(f_h^e(x), T^*)] = 0. \tag{6}$$

The procedure starts from the conservative assumption that the hold logic will be generated by simply mapping the BDD of f_h^e onto a network of multiplexors [9], [10]. This straightforward implementation can be obtained from the BDD of f_h^e in $O(N_{f_h^e})$ time [9], where $N_{f_h^e}$ is the number of nodes in the BDD on which variable reordering has been applied with the purpose of reducing its size. Obviously, the network obtained by direct BDD mapping is highly unoptimized. Therefore, its performance can be sensibly improved by standard logic optimization.

Under the assumption of a multiplexor-based implementation of the hold circuit, the longest path in the BDD gives us an estimate of the critical path for the hold network. Clearly, this is only a first order estimate, since it neglects two factors: the output load on a multiplexor and the load on the control inputs of the multiplexors. If the BDD is very "wide" in the lower levels (i.e., there are many nodes marked with variables which are at the bottom of the global order), the speed of the multiplexor-based network could be limited by the excessive load on the control inputs of the multiplexors. Similarly, if a node in the BDD is shared by many subtrees, the corresponding multiplexor has a large fan-out and its speed decreases. However, buffering can mitigate the problem and reduce the delay penalty in both situations. Our approach is to focus first on the number of levels of logic in the multiplexor-based network.

The algorithm for the constrained generation of f_h^e consists of two steps. First, the BDD of f_h is traversed and levelized: each node is marked with its *level*, that is, the length of the longest path between the node and the root of the BDD. Second, the constraint on the maximum number of levels is enforced. Let $d_{\text{mux}}(D_{\text{avg}}, C_{\text{avg}})$ be the delay of a multiplexor with a fan-out load of C_{avg} and an input drive D_{avg}, where C_{avg} and D_{avg} are two constants representing the expected average load on a multiplexor and the expected driving strength on its inputs. The maximum number of levels of logic allowed in the multiplexor network is given by

$$\Lambda_{\max} = \lfloor K_t T^* / d_{\text{mux}}(D_{\text{avg}}, C_{\text{avg}}) \rfloor \tag{7}$$

where K_t is a scaling constant that factors the expected effect of logic synthesis and optimization on the multiplexor network ($K_t < 1$ produces conservative results).

Starting from the nodes marked with higher level, the BDD is traversed, and all nodes for which $LEVEL > \Lambda_{\max}$ are replaced by the constant 1. This operation yields a function $f_h^e > f_h$. Notice that node elimination implies the reduction of the number of paths in the BDD longer than $\Lambda_{\max}$. In particular, elimination of a single node may cause a length reduction for an exponential number of paths.

We call *supersetting* the operator that eliminates a given node from a BDD, since it is the dual of the *subsetting* transformation proposed by Ravi and Somenzi in [11] in the context of reachability analysis of large finite-state machines. We have implemented the supersetting operator using a recursive procedure, described in [12], which is structured as most of the basic BDD operators. Supersetting techniques alternative to ours have been proposed in the recent literature. The interested reader may refer to [13] and [10] for more details on this subject.

One issue that still needs to be addressed concerns timing constraint violations due to nodes with excessive fan-out or excessively loaded input signals. Supersetting can be exploited again to eliminate such violations. Simple heuristics for marking nodes that would generate heavily loaded multiplexors, or for reducing the load on the inputs have been devised and are not described here because they do not add much to the understanding of the algorithms. In addition, these heuristics have a very marginal impact on the quality of the results, since the modification of the BDD's for reducing their depth almost always yields implementations that satisfy the timing constraint T^*.

Fig. 3 outlines the pseudocode of the BDD-based algorithm for the synthesis and optimization of f_h^e.

Procedure `Bdd2Logic` takes, as inputs, the original function f_h, the desired cycle time T^*, and the area bound

```
procedure Bdd2Logic (f_h, T*, A*) {
  f_h^e = VariableReordering(f_h);
  while(1) {
    C_{f_h^e} = Bdd2Network(f_h^e);
    C_{f_h^e} = LogicOptimization(C_{f_h^e});
    C_{f_h^e} = TechMapping(C_{f_h^e});
    AT_{C_{f_h^e}} = AddTimingAnalysis(C_{f_h^e});
    if ((MaxVal(AT_{C_{f_h^e}} < T*) and Area(C_{f_h^e}) ≤ A*)) return (C_{f_h^e});
    Λ_max = FindLambdaMax(f_h^e);
    LEVEL[] = Levelize(f_h^e);
    NodeListLev = MarkNodesLev(f_h^e, LEVEL[], Λ_max);
    f_h^e = Supersetting(f_h^e, NodeListLev);
    NodeListLoad = MarkNodesLoad(f_h^e, MuxLoad);
    f_h^e = Supersetting(f_h^e, NodeListLoad);
  }
}
```

Fig. 3. The `Bdd2Logic` algorithm.

A^* for the hold circuit, and it returns the hold circuit $C_{f_h^e}$ implementing an f_h^e which satisfies the required constraints.

The size (i.e., the number of nodes) of the BDD for f_h is first reduced through variable reordering, and the corresponding BDD is synthesized as a multiplexer-based logic network and subsequently optimized and mapped. The arrival time ADD $AT_{C_{f_h^e}}$ for the hold logic is computed. If both the timing and the area constraints are met by the implementation $C_{f_h^e}$ of f_h^e, such implementation is returned. Otherwise, a modification of f_h^e is required, following the supersetting paradigm presented earlier in this section.

After computing $\Lambda_{\max}$, the BDD for f_h^e is levelized and supersetting is first used to reduce the depth of the BDD and eventually to eliminate nodes possibly responsible for timing violations due to excessive load. At this point, the whole sequence of operations starts over.

Notice that procedure `Bdd2Logic` is guaranteed to terminate because supersetting eliminates at least one node in the BDD at each iteration. In all the cases we examined, one iteration was sufficient to find an implementation of f_h^e that satisfied the timing constraint T^*. In the worst case, the procedure terminates in a number of iterations which is proportional to the size of the BDD of f_h.

B. SOP-Based Heuristics

The main advantage of the BDD-based heuristics for the generation of the hold circuit lies in its well-controlled complexity. If the timing analysis tool can compute the hold function f_h, all steps for the generation of f_h^e require linear time in the size of its BDD. The major limitation of the BDD-based approach is that the hold circuit is a multiplexor network obtained from the BDD. Since speed is the primary requirement for the hold circuit, we may need to apply logic optimization to obtain a fast implementation. Most speed-up algorithms perform some form of *flattening* of the initial specification in order to resynthesize a faster network. Flattening transforms the initial specification into a two-level sum-of-products form. Unfortunately, when flattening a multiplexer network, the number of products in the flattened implementation may be exponentially larger than the number of multiplexers in the initial specification.

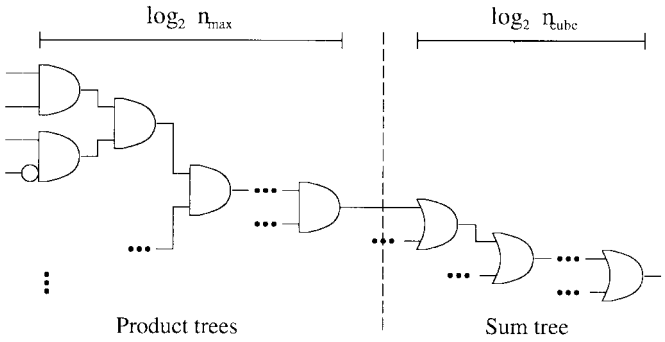


Fig. 4. General multilevel implementation of a SOP.

This implies an inherent difficulty in obtaining a fast implementation of the hold circuit, since our BDD-based algorithm produces a small multiplexer network that might be actually very hard to speed-up by logic optimization. In this section, we propose a heuristics for the synthesis of the hold circuit that, starting from the BDD of f_h, it first produces a sum-of-products description of f_h^e and then resorts to logic optimization to obtain a fast and compact gate-level netlist. In other words, we directly generate a flattened version of f_h^e from the BDD, without generating the multiplexor network.

Since the hold function is subject to the constraint that the delay of its final implementation must be shorter than T^*, the SOP generation procedure is delay-driven. Assume that a Boolean function with N inputs has been specified as a SOP, using just the AND, OR, and NOT operators, and that it contains n_{cube} product terms. The largest cube (i.e., product term) has $n_{\max}$ specified literals, where $n_{\max} \leq N$. If we neglect for simplicity the effect of the input loads (i.e., we assume infinite input driving strengths), it is easy to realize that the function can be implemented by a multilevel network having the structure shown in Fig. 4. The delay of such network is given by

$$d_{\text{sop}} = K_s \cdot \lceil log_2 n_{\text{cube}} \rceil + K_p \lceil log_2 n_{\max} \rceil + K_{in}. \quad (8)$$

The first term in the expression accounts for the delay through the balanced tree of two-input OR operators needed to implement the logic sum of n_{cube} cubes. The second term accounts for the delay through the balanced tree of two-input AND operators needed to implement the cube with the maximum number of specified literals. The last term accounts for the delay of a NOT operator (needed to complement the input variables, if they appear in negative phase in the cubes). Constants K_s, K_p, K_{in} represent the delay through the AND, OR, and NOT gates with unit load.

We cannot guarantee logarithmic delay for f_h, since it is well known that there exist functions which can be represented only with an exponential number of cubes in SOP form. Fortunately, the hold circuit can implement any function $f_h^e \geq f_h$. Consequently, we do not generate a cover for the hold function, but rather for its *complement* $f_h^{e'}$. We enumerate cubes of f_h' and we include them in a partial cover. When the enumeration is stopped (by a stopping criterion discussed later) the procedure returns a partial cover of f_h': $f_h^{e'} \leq f_h'$. If we implement the partial cover and we complement its value, we

```
procedure Bdd2Cover (f', n_cube, n_max) {
    CubeList = ∅;
    foreach (Cube ∈ f') {
        if (NLits(Cube) ≤ n_max)
            InsertInOrder(Cube, cubeList);
        if ( Size(CubeList) > n_cube)
            RemoveLast(CubeList);
    }
    return (CubeList);
}
```

Fig. 5. The `Bdd2Cover` algorithm.

obtain an implementation of $f_h^e \geq f_h$, thereby achieving our original objective. The pseudocode for the cover generation algorithm is shown in Fig. 5.

The procedure receives, as input parameters, the BDD representation of f_h', the bound on the total number of cubes n_{cube}, and the bound on the maximum number of specified literals in any considered cube $n_{\max}$. The choice of the values n_{cube} and $n_{\max}$ is driven by the required timing constraints. More specifically, the constraint T^* is split into two contributions: $T_{\text{prod}} = T^* \cdot D_{\text{prod}}$ and $T_{\text{sum}} = T^* \cdot D_{\text{sum}}$. The coefficients D_{prod} and D_{sum} represent the fraction of the total time to be spent in computing the logic products and the fraction of the time to be spent in computing the logic sum, respectively. Usually, $D_{\text{prod}} < D_{\text{sum}}$ because the number of cubes is much larger than the number of literals in a cube. From T_{sum}, we compute $n_{\text{cube}} = \lfloor 2^{T_{\text{sum}}/K_s} \rfloor$. Similarly, T_{prod} is used to compute $n_{\max} = \lfloor 2^{T_{\text{prod}}/K_p} \rfloor$.

The algorithm consists of an outer loop for cube generation. Whenever a newly generated cube is examined, it is inserted in the selected cube list if and only if the number of specified literals (i.e., literals appearing in the cube in either directed or complemented form) is smaller than or equal to $n_{\max}$. Otherwise, the cube is simply dropped. The list is kept in decreasing order: large cubes are inserted at the top of the list (a large cube has a small number of specified literals). If the number of cubes in the list becomes larger than n_{cube}, the cube at the bottom of the list is discarded. Upon completion of the loop, the list is returned. It contains the n_{cube} largest cubes of f_h produced during cube enumeration.

It should be observed that procedure `Bdd2Cover` requires the explicit enumeration of all cubes obtained from the BDD representation. Consequently, the main shortcoming of this simple algorithm is its worst case exponential number of iterations. A straightforward solution is to limit the number of iterations of the **foreach** loop to a user-defined upper bound. In this case, we cannot guarantee that the list will contain the largest cubes in the cover, but only the largest cubes generated during the selected number of iterations.

Given the list of cubes, the final implementation of $f_h^{e'}$ is obtained through logic optimization. The hold circuit is then realized by simply inserting an inverter at the output. Since we start from a SOP specification, the optimization procedures are more effective in finding fast implementations with small area, than in the case of multiplexer-based network, as confirmed by the experimental results.

C. Practical Issues

An implicit assumption made throughout the paper is that the presence of the hold circuit does not perturb the timing behavior of the original logic of the unit. Unfortunately, in principle this is not true. Although the combinational logic implementing the f_h^e is never shared with the logic of the original circuit, they are both driven by the same inputs. When we add the hold circuit in the telescopic unit, the load on the flip-flops at the inputs of the stage increases, and the propagation delay increases as well. As a consequence, the timing in the original circuit may change. In particular, paths with propagation delay originally below T^* may become too slow and violate the timing requirements. If this happens, the telescopic unit may malfunction, because the hold circuit is not guaranteed to be active for the paths that have become too slow due to its presence.

To tackle this problem we have devised two strategies. The first and more conservative approach specifies an additional load on the flip-flop outputs (i.e., the inputs of the combinational logic) when performing the initial timing analysis of the stage. This can be done by connecting an additional gate (i.e., a buffer) to each flip-flop output. In the final implementation of the telescopic unit, the additional gates are the input drivers for the hold circuit. The addition of the drivers allows us to decouple the timing analysis of the combinational logic from the synthesis of the hold circuit. The strategy is conservative since it assumes that an additional driver is needed to drive every input of the hold circuit.

A more aggressive alternative assumes that the presence of the hold logic does not sensibly change the timing in the original logic. More specifically, it assumes that none of the paths in the combinational block which are not covered by f_h^e becomes slower than T^*. Function f_h^e is thus synthesized and wired to the original stage in the usual way. Timing analysis is then performed: if some violation is detected (i.e., paths longer than T^* in the original logic are activated by input vectors in the OFF-set of f_h^e), the hold circuit is resynthesized using a new, artificially reduced $T_{\text{red}}^* = T^* - T_{\text{viol}}$. The decrease T_{viol} equals the maximum violation that occurs in the original circuit when the hold circuit is inserted. The process is iterated until the addition of the hold circuit no longer originates a violation.

Although the second alternative may seem more risky and computationally expensive, we have empirically observed that often the insertion of the hold circuit does not create any violation, and the blind addition of buffers may thus be an overkill. In our experiments, we have chosen the second approach with good success.

V. Controller Design

In all practical cases, computational units are embedded in a larger system and must be interfaced to the environment in a consistent and correct fashion. In this section, we show how to incrementally modify the controller of a data-path when the latter is transformed into a telescopic unit.

For the sake of illustration, consider the following design scenario. The behavioral description of a system is provided by

the designer. Behavioral synthesis is performed and an initial implementation, consisting of a controller and a data-path, is obtained. The designer (or a high-level design exploration tool) examines the data-path and decides which unit is to be transformed into a telescopic unit. For instance, the slowest unit in the data-path (which dominates the cycle time) can be chosen. Thus, the system can potentially run with a cycle time $T^* < T$. The introduction of a telescopic unit implies the existence of a new signal f_h. Such signal is an input to the controller that must be modified (i.e., resynthesized) to take into account the variable latency of the telescopic unit. Controller resynthesis must satisfy two key requirements:

1) it must guarantee functional correctness;
2) the complete system (i.e., controller and data-path) must run at the new cycle time T^*.

We briefly describe a controller transformation procedure for data-paths containing telescopic units. A complete treatment of this topic can be found in [14]. Numerous controller generation algorithms for behavioral synthesis have been presented in the past [15]. Most synthesis techniques generate controller representations in terms of state tables (or equivalent formalisms) of a *finite-state machine* (FSM) model of the control unit. Other techniques [16] generate a network of simple state machines, each controlling a task or a set of concurrent tasks and where transitions are triggered by task *completion* signals. Since the complement of the hold signal denotes the completion of a computation, such schemes are easily adapted to support telescopic units. Unfortunately, the implementations may be inefficient in terms of area utilization and may become impractical to control data-paths with large sets of tasks.

We consider here the modifications needed to be applied to a state table representation of a control unit in order to control telescopic units. Thus, this procedure is compatible with most current behavioral synthesis methodologies. Before describing the procedure in detail, we define two types of controller outputs (also called control signals), namely, the *load signals* and the *steering signals*. We call *load signals* ld_i the FSM outputs that control the loading of new values into the data-path registers. A load signal is *active* when it allows the overwriting of the old data in the register. When the load signal is inactive, the register holds its value. Without loss of generality, we assume that the active value is $ld_i = 1$. We call *steering signals* m_i the FSM outputs that control the multiplexors in the data path. Such multiplexors implement the steering logic: at the input of a unit they are used to select the operands, while at the output, they select where to store the result of a computation. In the state table of the controller, a state is associated with each control step, and the edge leaving the state is labeled with the values of the load and the steering signals.

Assume that the fixed-latency unit U_i is transformed into a telescopic one with hold function f_h. If unit U_i is active in state S_j, the controller's state table is modified applying the following rules.

- The unconditional transition from state S_j to state S_{j+1} is now conditioned by the event $f_h = 0$. The output fields in the transition are left unchanged. In other words, if the telescopic unit requires just one cycle to complete, the system can move to the next control step.
- A new state SH_j is added.[2] The FSM transitions from state S_j to state SH_j when $f_h = 1$. The load signals in the transition are all inactive, while the steering signals have the same value as in the transition from S_j to S_{j+1}. This transition is taken when the telescopic unit takes one additional cycle to complete. If this is the case, the registers at the inputs and outputs of the unit must not be reloaded because the computation has not terminated. Clearly, the steering signals must not be changed because the input operands of the unit must be held constant.
- An unconditional transition from state SH_j to state S_{j+1} is added. The outputs for the transition are exactly the same as the ones in the transition from S_j to S_{j+1}. The additional transition is taken when the computation of the telescopic unit takes two cycles. Notice that the value of f_h does not need to be sampled because, by construction, the unit completes its execution in either one or two cycles, but not more.

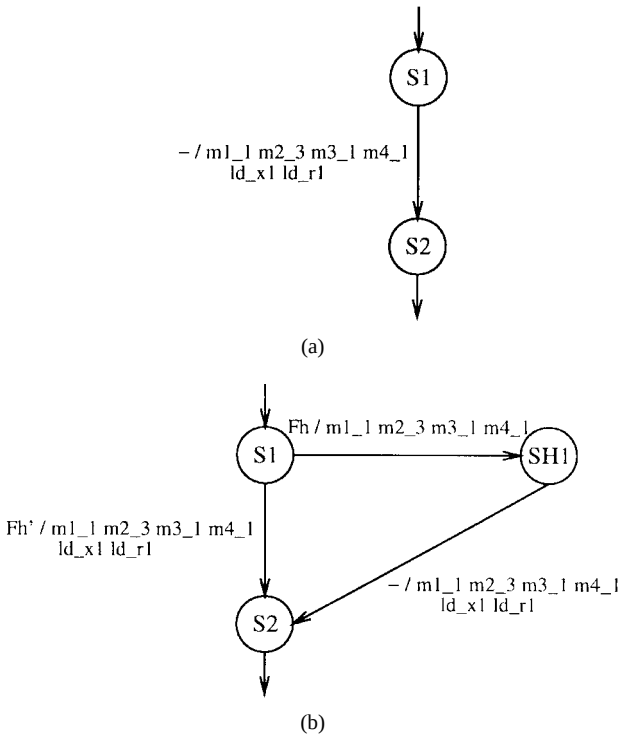


Fig. 6. (a) Fragments of fixed-latency controller and (b) transformed (variable-latency) controller.

Example 2: Consider the controller fragment shown in Fig. 6(a). Assume that a unit scheduled in state $S1$ becomes a telescopic one. The transformation of the controller for state $S1$ is shown in Fig. 6(b). Output signals `ld_x1` and `ld_r1` are load signals, while outputs `m1_1`, `m2_2`, `m3_1`, and `m4_1` are steering signals. Observe that one state has been added

[2] We are disregarding conditionals for the sake of simplicity. In the presence of conditionals, a state S can have multiple out-going edges. In this case, a new state SH_k should be added for each out-going edge k.

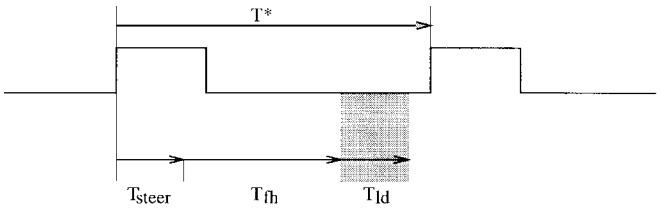


Fig. 7. Timing diagram of the interaction between f_h and the control logic.

(SH_1). Moreover, transitions in the transformed controller depend on f_h, the hold function of the telescopic unit.

In the worst case, i.e., when the telescopic unit is active in all control steps, the number of states in the control FSM increases by a factor of two. More in general, if multiple telescopic units are instantiated, the increase in the number of states is exponential in the number of telescopic units. Thus, designs with many telescopic units should adopt a distributed control-generation style [16], where the controller is implemented as a network of small interacting FSM's.

When a single telescopic unit is instantiated in the data-path, the complexity increase in the controller is well controlled. The number of states remains linear in the number of control steps, and the number of input signals increases linearly with the number of telescopic units. Hence, the increase in area of the controller is not a serious concern (the total area is still dominated by the data path). Unfortunately, this is not the case for timing.

Fig. 7 shows the timing diagram for the interaction between telescopic unit and controller. Delay T_{steer} is the time required by the controller for setting stable values on the input multiplexors of the telescopic unit. Such delay must be taken into account even for fixed-latency units. Delay T_{f_h} is the time required by f_h to settle. Delay T_{ld} is the time required by the load signals of the controller to reach the stable value after f_h has settled. The path with delay $T_{\text{steer}} + T_{f_h} + T_{ld}$ is exercised when, for example, the telescopic unit is fed with a pattern that requires two cycles immediately after a pattern requiring a single cycle. Checking for correct timing requires to verify that $T_{\text{steer}} + T_{f_h} + T_{ld} < T^*$. Obviously, this condition implies tighter timing constraints for the hold circuit: $T_{f_h} < T^* - T_{\text{steer}} - T_{ld}$.

Another important timing-related issue is the presence of glitches on the steering signals when the controller's FSM transitions from a S state to one of the new SH states. Although the final value of the steering signals is unchanged, a glitch during the transition may cause spurious transitions on the telescopic unit's inputs while the unit is still completing its computation. Propagation of such spurious transitions may cause an increase in the time needed for the unit's outputs to settle. Hence, the gate-level implementation of the steering signals should be glitch-free for all transitions from S states to SH states. Glitch-free synthesis techniques [17] can be used to satisfy this requirement.

VI. Experimental Results

We have implemented procedures `Bdd2Logic` and `Bdd2Cover`, as well as the surrounding software as an extension of SIS [18] using CUDD [10] as the underlying BDD/ADD package. Experiments have been run on a DEC AXP 1000/400 with 256 MB of memory.

We present two sets of data. The first one concerns the use of telescopic units as a pure throughput optimization technique. The second set shows the applicability of telescopic units for area optimization under throughput constraints.

A. Throughput Optimization

We have considered all large (>100 gates) benchmarks in the MCNC'91 combinational multilevel suite [19] (53 examples). The circuits have been first optimized for speed using a modified version of the `script.delay` SIS script, in which the `full_simplify -l`, and sometimes the `red_removal` commands have been removed to allow the optimization to complete on the large examples. Then, they have been mapped for speed with either the `map -n1 -AFG` or the `map -m1` command onto a cell library containing inverters, buffers, and two-input NAND and NOR gates. The unit gate delay model has been adopted for the ADD-based timing analysis.

1) BDD-Based Synthesis Procedure: We have run the BDD-based synthesis procedure on the delay-optimized circuits trying to obtain maximum-throughput telescopic units. To accomplish this task we have specified several decreasing values for T^*, and we have synthesized the hold circuit until we have found a value for which a further cycle time reduction caused a decrease in throughput (due to the high probability of the hold function).

For 43 examples, the use of telescopic units has produced a substantial throughput improvement. On the other hand, in five cases (circuits `C499`, `i3`, `i4`, `i6`, and `i7`), the throughput did not increase. The reason for the failure is due to the delay distribution in the circuits. For example, the critical path delay of `i6` is $T = 6$ time units. If we specify $T^* = 5$ and we extract f_h, we obtain $\text{Prob}(f_h) = 1$. Finally, in five cases (circuits `C1355`, `C2670`, `C3540`, `C6288`, and `i10`), the ADD-based timing analysis did not complete, due to the size of either the BDD's of the output functions of the unit or the arrival time ADD's to be constructed. Thus, our tool could not proceed to the generation of f_h.

Table I reports the data for the 43 examples on which throughput optimization has succeeded. Benchmarks are sorted by increasing size. Columns *Circuit*, *In*, *Out*, *Gt*, T, and P give the name, the number of inputs, outputs, and gates, the true delay, and the throughput of the original circuit, respectively. Column $\text{Prob}(f_h^e)$ shows the probability of f_h^e, column Gt^* gives the total number of gates of the telescopic unit, column T^* reports the cycle time at which the telescopic unit is clocked to achieve the increased throughput of column P^*, and column $T(f_h^e)$ tells the arrival time of the hold signal. Columns ΔP and ΔGt give the percentage of throughput improvement and area overhead (in terms of gates) of the telescopic unit. Finally, column *Time* reports the central processing unit (CPU) time, in seconds, required to perform the ADD-based timing analysis, as well as the synthesis and the optimization of f_h^e for a given T^*.

TABLE I
THROUGHPUT OPTIMIZATION USING THE BDD-BASED SYNTHESIS PROCEDURE

Circuit	*In*	*Out*	*Gt*	T	P	$Prob(f_h^e)$	Gt^*	T^*	P^*	$T(f_h^e)$	ΔP	ΔGt	*Time*
pcler8	27	17	105	12	0.0833	0.18750	109	7	0.1295	3	55.4%	3.8%	0.1
mux	21	1	106	14	0.0714	0.05070	145	12	0.0812	9	13.7%	36.8%	0.3
cordic	23	2	126	15	0.0667	0.05270	153	11	0.0885	10	32.8%	21.4%	0.2
frg1	28	3	143	15	0.0667	0.37500	145	8	0.1015	2	52.3%	0.1%	1.6
sct	19	15	143	8	0.1250	0.04680	152	6	0.1628	4	30.2%	6.3%	0.1
unreg	36	16	147	6	0.1667	0.25000	149	3	0.2917	2	75.0%	1.4%	0.1
b9	41	21	150	11	0.0909	0.46000	161	7	0.1100	6	21.0%	7.3%	0.2
f51m	8	8	152	11	0.0909	0.10900	165	10	0.0946	7	4.0%	8.5%	0.2
comp	32	3	174	21	0.0476	0.00366	220	19	0.0525	9	10.3%	26.4%	0.6
lal	26	19	179	10	0.1000	0.00195	191	9	0.1110	5	11.0%	6.7%	0.2
count	35	16	205	12	0.0833	0.25000	207	6	0.1458	2	75.0%	0.9%	0.2
cht	47	36	209	6	0.1666	0.25000	211	5	0.1750	2	5.0%	0.9%	0.1
c8	28	18	211	9	0.1111	0.10900	218	7	0.1351	3	21.6%	3.3%	0.2
my_adder	33	17	225	34	0.0294	0.13600	286	18	0.0518	9	76.0%	27.1%	1.6
i2	201	1	242	12	0.0833	0.19500	256	8	0.1128	7	35.4%	5.8%	1.7
term1	34	10	242	17	0.0588	0.10232	272	11	0.0862	10	46.5%	12.3%	0.9
9symml	9	1	252	14	0.0714	0.03700	303	13	0.0755	9	5.7%	20.2%	0.9
apex7	49	37	302	16	0.0625	0.37100	323	10	0.0815	9	30.3%	6.9%	0.6
ttt2	24	21	306	10	0.1000	0.12100	322	8	0.1174	7	17.4%	5.2%	1.0
example2	85	66	358	13	0.0769	0.08100	382	10	0.0960	9	24.7%	6.7%	0.9
C432	36	7	404	27	0.0370	0.00052	435	26	0.0385	10	3.8%	7.7%	12.3
too_large	38	3	417	21	0.0476	0.01921	462	17	0.0582	7	22.3%	10.7%	6.1
i5	133	66	445	12	0.0833	0.03300	476	10	0.0984	9	18.0%	6.9%	1.8
x1	51	35	452	15	0.0667	0.08440	491	10	0.0958	8	43.7%	8.6%	0.8
x4	94	71	498	12	0.0833	0.16682	539	10	0.0916	9	9.9%	8.2%	1.5
C880	60	26	541	30	0.0333	0.02704	595	21	0.0469	20	40.8%	9.9%	361.5
alu2	10	6	783	35	0.0285	0.39453	810	19	0.0422	10	47.8%	3.4%	2.7
i9	88	63	813	16	0.0625	0.02734	828	14	0.0704	6	12.7%	1.8%	4.6
rot	135	107	840	23	0.0434	0.20955	937	19	0.0471	18	8.5%	11.5%	97.0
x3	135	99	872	14	0.0714	0.02168	928	11	0.0899	10	25.9%	6.4%	1.2
apex6	135	99	889	17	0.0588	0.00020	905	16	0.0625	10	6.2%	1.8%	1.1
t481	16	1	1043	22	0.0454	0.16462	1151	18	0.0509	17	12.1%	10.3%	7.6
frg2	143	139	1048	16	0.0625	0.25000	1054	9	0.0972	2	55.5%	0.5%	5.6
C1908	33	25	1132	26	0.0384	0.08404	1224	21	0.0456	20	18.7%	8.1%	76.1
dalu	75	16	1316	23	0.0434	0.37329	1394	16	0.0508	6	16.9%	5.9%	9.9
vda	17	39	1416	12	0.0833	0.02680	1447	11	0.0897	10	7.6%	2.2%	1.8
alu4	14	8	1457	39	0.0256	0.35992	1520	20	0.0410	17	59.9%	4.3%	8.9
i8	133	81	1485	16	0.0625	0.06884	1519	13	0.0742	11	18.8%	2.2%	5.3
pair	173	137	1956	28	0.0357	0.00830	2027	24	0.0415	18	16.1%	3.6%	11.4
C5315	178	123	3079	41	0.0243	0.25000	3099	25	0.0350	9	44.0%	0.6%	1449.0
k2	45	45	2393	18	0.0555	0.08120	2571	16	0.0599	15	7.9%	7.4%	10.0
C7552	207	108	5031	30	0.0333	0.01226	5228	23	0.0432	19	29.7%	3.9%	161.8
des	256	245	5084	27	0.0370	0.01562	5119	24	0.0413	7	11.6%	0.6%	12.2
Average											27.5%	7.7%	

In all cases, we have obtained a noticeable average throughput increase (27.5% on average) with a limited area overhead (7.7% on average). It is important to observe that the speed optimization for the initial circuits has been pushed all the way to the limit; therefore, the throughput increase achieved on each example can be totally awarded to the use of telescopic units. In some cases the optimization could have been even more aggressive than what we implemented by choosing $T^* < T/2$.

2) SOP-Based Synthesis Procedure: In order to check the effectiveness of the SOP-based heuristics, we have chosen those circuits (out of the 43 considered before) where either the throughput improvement was smaller than 10%, or the area penalty was larger than 10%. A total of 16 examples has thus been selected from Table I. The SOP-based procedure has been run on the reference versions of such examples for heavy-duty optimization of f_h^e. Table II compares, for each circuit, the results obtained with the BDD-based and the SOP-based synthesis procedures.

Our primary interest was the evaluation of the impact of the SOP-based heuristics on the area of the telescopic units. However, in order to make the comparison of the two heuristics as fair as possible, we have not allowed any throughput degradation with respect to the units obtained through the BDD-based procedure. In addition, for each example, we have kept T^* to the value used for the BDD-based synthesis. This is for better identifying the effects of the synthesis heuristics on the implementation of the hold circuit.

The results of the comparison are in favor of the SOP-based approach by an amount which goes beyond our expectations.

TABLE II
THROUGHPUT OPTIMIZATION USING THE SOP-BASED SYNTHESIS PROCEDURE

Circuit	*In*	*Out*	*Gt*	*T*	*P*	*Heur*	$Prob(f_h^e)$	Gt^*	T^*	P^*	$T(f_h^e)$	ΔP	ΔGt	*Time*
mux	21	1	106	14	0.0714	BDD	0.05070	145	12	0.0812	9	13.7%	36.8%	0.3
						SOP	0.05070	143	12	0.0812	9	13.7%	34.9%	1.0
cordic	23	2	126	15	0.0667	BDD	0.05270	153	11	0.0885	10	32.8%	21.4%	0.2
						SOP	0.02636	148	11	0.0897	10	34.5%	17.5%	1.2
f51m	8	8	152	11	0.0909	BDD	0.10900	165	10	0.0946	7	4.0%	8.5%	0.2
						SOP	0.10900	165	10	0.0946	7	4.0%	8.5%	0.2
comp	32	3	174	21	0.0476	BDD	0.00366	220	19	0.0525	9	10.3%	26.4%	0.6
						SOP	0.00005	216	19	0.0526	9	10.5%	24.1%	136.7
cht	47	36	209	6	0.1666	BDD	0.25000	211	5	0.1750	2	5.0%	0.9%	0.1
						SOP	0.25000	211	5	0.1750	2	5.0%	0.9%	0.3
my_adder	33	17	225	34	0.0294	BDD	0.13600	286	18	0.0518	9	76.0%	27.1%	1.6
						SOP	0.03515	261	18	0.0545	7	85.6%	16.0%	112.1
term1	34	10	242	17	0.0588	BDD	0.10232	272	11	0.0862	10	46.5%	12.3%	0.9
						SOP	0.10232	272	11	0.0862	10	46.5%	12.3%	28.9
9symml	9	1	252	14	0.0714	BDD	0.03700	303	13	0.0755	9	5.7%	20.2%	0.9
						SOP	0.03610	290	13	0.0755	9	5.7%	15.1%	0.8
C432	36	7	404	27	0.0370	BDD	0.00052	435	26	0.0385	10	3.8%	7.7%	12.3
						SOP	0.00052	414	26	0.0385	9	3.8%	2.5%	12.2
too_large	38	3	417	21	0.0476	BDD	0.01921	462	17	0.0582	7	22.3%	10.7%	6.1
						SOP	0.01921	462	17	0.0582	7	22.3%	10.7%	125.1
x4	94	71	498	12	0.0833	BDD	0.16682	539	10	0.0916	9	9.9%	8.2%	1.5
						SOP	0.02434	510	10	0.0987	8	18.5%	2.4%	132.6
rot	135	107	840	23	0.0434	BDD	0.20955	937	19	0.0471	18	8.5%	11.5%	97.0
						SOP	0.15125	916	19	0.0486	16	12.0%	9.1%	204.6
apex6	135	99	889	17	0.0588	BDD	0.00020	905	16	0.0625	10	6.2%	1.8%	1.1
						SOP	0.00020	905	16	0.0625	10	6.2%	1.8%	2.7
t481	16	1	1043	22	0.0454	BDD	0.16462	1151	18	0.0509	17	12.1%	10.3%	7.6
						SOP	0.10426	1137	18	0.0526	16	15.8%	9.0%	80.1
vda	17	39	1416	12	0.0833	BDD	0.02680	1447	11	0.0897	10	7.6%	2.2%	1.8
						SOP	0.01916	1430	11	0.0901	9	8.1%	1.0%	2.8
k2	45	45	2393	18	0.0555	BDD	0.08120	2571	16	0.0599	15	7.9%	7.4%	10.0
						SOP	0.08120	2558	16	0.0599	15	7.9%	6.8%	22.6
Average (BDD)												17.0%	13.4%	
Average (SOP)												18.8%	10.8%	

In fact, not only the average area overhead has decreased from 13.4 to 10.8%, but a further average throughput increase from 17.0 to 18.8% has been achieved as a by-product. In a few cases, the worst-case delay of the hold circuit has also decreased.

As expected, the computation time of the SOP-based heuristics is higher than that of the BDD-based one. Even though in most of the cases the difference in running time is negligible, there are examples where the SOP-based synthesis has required a few minutes to complete.

B. Area Optimization

For this set of experiments, the initial circuits have been optimized for area by iteratively applying the `script.rugged` SIS script (without the `full_simplify -m nocomp`) followed by the `red_removal` command (whenever this has been possible), and mapped for area using the `map -m0` command onto the usual cell library. Then, a 20% throughput optimization has been targeted using two different approaches: first, by transforming the circuits into telescopic units using the BDD-based heuristics; second, by optimizing the circuits for delay using SIS. Table III reports the experimental data. (Examples are sorted as in Tables I and II). In particular, columns *Original–Gt*, *Original–T*, and *Original–P* report the number of gates, the cycle time, and the average throughput of the original (minimum area) circuits. Columns *Telescopic Unit–Gt* and *Telescopic Unit–*ΔGt give the number of gates and the percentage of gates overhead required by the telescopic units to produce a 20% throughput increase. Columns *Delay Optimized*–Gt and *Delay Optimized–*ΔGt show similar data for the delay optimized circuits. Finally, the right-most column indicates area wins $(+)$ and losses $(-)$ of the telescopic units over the delay optimized examples.

Due to the difficulty of exactly controlling the throughput increase, a $\pm 2.5\%$ slack has been allowed. The symbol — in a column indicates that the desired throughput improvement could not be obtained. This situation has occurred in one case only for the telescopic units (circuit `C432`) and in 19 cases for the delay optimized circuits. It should be observed that the telescopic units have out-performed the delay optimized circuits, in terms of gate count, in the majority of the cases (36 examples out of 43). Only on benchmark `C432` both optimizations failed. On average, the area overhead due to the use of telescopic units has been around 16.5%, while for the delay optimized circuits it has been approximately 30.5%. Such average is obviously computed only for the examples on which both optimizations have succeeded.

TABLE III
AREA OPTIMIZATION UNDER THROUGHPUT CONSTRAINTS

Circuit	*Original*			*Telescopic Unit*		*Delay Optimized*		
	Gt	T	P	Gt	ΔGt	Gt	ΔGt	
pcler8	101	14	0.0714	109	7.9%	—	—	+
mux	69	15	0.0666	72	4.3%	—	—	+
cordic	77	19	0.0526	98	27.2%	126	63.6%	+
frg1	124	21	0.0476	134	8.0%	143	15.3%	+
sct	76	18	0.5555	89	17.1%	102	34.2%	+
unreg	118	8	0.1250	149	26.2%	147	24.5%	−
b9	128	13	0.0769	142	10.9%	—	—	+
f51m	68	23	0.0333	117	72.0%	141	107.3%	+
comp	123	23	0.0434	146	21.1%	—	—	+
lal	105	16	0.0625	116	10.4%	120	14.2%	+
count	138	21	0.0476	140	1.4%	157	13.7%	+
cht	179	8	0.1250	180	0.5%	209	16.7%	+
c8	141	13	0.0769	143	1.4%	156	10.6%	+
my_adder	190	37	0.0270	263	38.4%	—	—	+
i2	220	14	0.0714	294	33.6%	—	—	+
term1	245	21	0.0476	317	29.3%	—	—	+
9symml	192	21	0.0476	271	41.1%	240	25.0%	−
apex7	238	19	0.0526	265	11.3%	—	—	+
ttt2	161	19	0.0526	186	15.5%	181	12.4%	−
example2	306	16	0.0625	315	2.9%	—	—	+
C432	179	30	0.0333	—	—	—	—	=
too_large	364	23	0.0434	411	12.9%	—	—	+
i5	198	18	0.5555	246	24.2%	388	95.9%	+
x1	324	16	0.0625	416	28.3%	—	—	+
x4	408	17	0.0588	430	5.3%	471	29.3%	+
C880	431	51	0.0196	460	6.7%	493	14.4%	+
alu2	460	42	0.0238	498	8.2%	—	—	+
i9	663	20	0.0500	672	1.3%	813	22.6%	+
rot	731	29	0.0344	854	16.8%	840	14.9%	−
x3	723	19	0.0526	788	8.9%	790	9.2%	+
apex6	769	21	0.0476	905	17.6%	889	15.6%	−
t481	729	27	0.0370	1047	43.6%	—	—	+
frg2	695	29	0.0344	720	3.5%	757	8.9%	+
C1908	498	42	0.0238	606	21.6%	778	56.8%	+
dalu	891	40	0.0250	962	7.9%	1003	12.6%	+
vda	570	19	0.0434	668	17.2%	1022	79.2%	+
alu4	699	44	0.0227	854	22.1%	—	—	+
i8	1074	19	0.0526	1141	6.2%	—	—	+
pair	1630	41	0.0243	1855	13.8%	1891	16.0%	+
C5315	1705	48	0.0208	1940	13.7%	—	—	+
k2	1087	27	0.0370	1431	31.6%	1301	19.6%	−
C7552	3061	36	0.0277	3378	10.3%	—	—	+
des	3668	31	0.0322	4216	14.9%	—	—	+

VII. CONCLUSIONS

We have presented a technique for the automatic generation of variable-latency, high-performance units that allow us to increase the performance of digital systems beyond the limits achievable by traditional logic optimization. Thanks to symbolic timing analysis, we identify the conditions for which a computation takes longer than the desired cycle time. We then generate a signal which communicates to the environment when the correct result is available. We also automatically modify the controller to properly handle the variable-latency units that are introduced in the system. Experimental results demonstrate that the technique represents a valuable alternative to existing throughput optimization approaches.

ACKNOWLEDGMENT

The authors wish to thank I. Bahar for helping them with the ADD-based timing analysis code, and the anonymous DAC-97 reviewers for their useful comments on paper [12].

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufmann, 1996.

[2] S. F. Oberman and M. J. Flynn, "Design issues in division and other floating-point operations," *IEEE Trans. Comput.*, vol. 46, pp. 154–161, Feb. 1997.
[3] M. A. Kishinevsky, *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. New York: Wiley, 1994.
[4] R. I. Bahar, H. Cho, G. D. Hachtel, E. Macii, and F. Somenzi, "Timing analysis of combinational circuits using ADD's," presented at the *EDTC-94: IEEE Eur. Design Test Conf.* , pp. 625–629, Paris, France, Feb. 1994.
[5] S. Hassoun and C. Ebeling, "Architectural retiming: Pipelining latency-constrained circuits," presented at the *DAC-33: ACM/IEEE Design Automation Conf.*, pp. 708–713, Las Vegas, NV, June 1996.
[6] S. Nowick, "Design of a low-latency asynchronous adder using speculative completion," *IEE Proc. Comput. Digital Techniques*, vol. 143, pp. 301–307, Sept. 1996.
[7] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, pp. 79–85, Aug. 1986.
[8] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebraic decision diagrams and their applications," *Formal Methods Syst. Design*, vol. 10, pp. 171–206, 1997.
[9] L. Burgun, N. Dictus, A. Greiner, E. Prado Lopes, and C. Sarwary, "Multilevel optimization of very high complexity circuits," presented at the *EuroDAC-94: IEEE Eur. Design Automation Conf.*, pp. 14–19, Grenoble, France, Sept. 1994.
[10] F. Somenzi, "CUDD: University of Colorado decision diagram package, release 2.1.2," Dept. ECE, Univ. Colorado, Boulder, Tech. Rep., Apr. 1997.
[11] K. Ravi and F. Somenzi, "High-density reachability analysis," presented at the *ICCAD-95: IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 154–158, San Jose, CA, Nov. 1995.
[12] L. Benini, E. Macii, and M. Poncino, "Telescopic units: Increasing the average throughput of pipelined designs by adaptive latency control," presented at the *DAC-34: ACM/IEEE Design Automation Conf.*, pp. 22–27, Anaheim, CA, June 1997.
[13] T. Shiple, "Formal analysis of synchronous circuits," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, 1996.
[14] L. Benini, E. Macii, and M. Poncino, "Efficient controller design for telescopic units," presented at the *ISIS-97: IEEE Int. Conf. Innovative Syst. in Silicon*, pp. 290–299, Austin, TX, Oct. 1997.
[15] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[16] D. Ku and G. De Micheli, *High-Level Synthesis of ASIC's Under Timing and Synchronization Constraints*. Norwell, MA: Kluwer, 1992.
[17] L. Lavagno and A. Sangiovanni-Vincentelli, *Algorithms for Synthesis and Testing of Asynchronous Circuits*. Norwell, MA: Kluwer, 1993.
[18] E. M. Sentovich, K. J. Singh, C. W. Moon, H. Savoij, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuits design using synthesis and optimization," presented at the *ICCD-92: IEEE Int. Conf. Comput. Design*, pp. 328–333, Cambridge, MA, Oct. 1992.
[19] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," MCNC: Microelectronics Center of North Carolina, Research Triangle Park, NC, Tech. Rep., Jan. 1991.

Luca Benini received the B.S. degree (*summa cum laude*) in electrical engineering from the University of Bologna, Italy, in 1991, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, CA, in 1994 and 1997, respectively.

Since 1997, he has been a Research Associate in the Department of Electronics and Computer Science at the University of Bologna and a Post-Doctoral Fellow in the Department of Electrical Engineering at Stanford University. He also holds a position as a Visiting Scientist at the Hewlett-Packard Laboratories, Palo Alto, CA. His research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications.

Dr. Benini has been a Member of the Technical Program Committee for the 1998 Design and Test in Europe Conference and International Symposium on Low Power Design.

Enrico Macii (M'92) received the Dr.Eng. degree in electrical engineering from Politecnico di Torino, Italy, in 1990, the Dr.Sc. degree in computer science from Università di Torino in 1991, and the Ph.D. degree in computer engineering from Politechnico di Torino in 1995.

From May 1991 through August 1991, he was Visiting Faculty at the University of California at Los Angeles, and from September 1991 through September 1994, he was Adjunct Faculty at the University of Colorado at Boulder. Currently he is an Assistant Professor of Electrical and Computer Engineering at Politecnico de Torino. His research interests include synthesis, verification, simulation, and testing of digital circuits and systems.

Massimo Poncino (M'97) received the Dr.Eng. degree in electrical engineering from Politecnico di Torino, Italy, in 1989, and the Ph.D. degree in computer engineering from Politecnico di Torino in 1993.

From March 1993 through May 1994, he was Visiting Faculty at the University of Colorado at Boulder. Currently he is an Assistant Professor of Electrical and Computer Engineering at Politecnico di Torino. His research interests include synthesis, verification, simulation, and testing of digital circuits and systems.

Giovanni De Micheli (F'94) received the Nuclear Engineer degree from Politecnico di Milano, Italy, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1980 and 1983.

He is Professor of Electrical Engineering, and by courtesy, of Computer Science at Stanford University, CA. Previously, he held positions at the IBM T. J. Watson Research Center, Yorktown Heights, NY, at the Department of Electronics of the Politecnico di Milano, Italy, and at Harris Semiconductor, Melbourne, FL. His research interests include several aspects of the computer-aided design of integrated circuits and systems, with particular emphasis on automated synthesis, optimization, and validation. He is the author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994); coauthor of *Dynamic Power Management: Circuit Techniques and CAD Tools* (Norwell, MA: Kluwer, 1998) and of *High-level Synthesis of ASIC's under Timing and Synchronization Constraints* (Norwell, MA: Kluwer, 1992); and co-editor of *Hardware/Software Co-design* (Norwell, MA: Kluwer, 1995) and of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation* (The Netherlands: Martinus Niijoff Publishers, 1986). He was also codirector of the NATO Advanced Study Institutes on Hardware/Software Co-design, held in Tremeezo, Italy, in 1995, and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, in 1986.

Dr. De Micheli was granted a Professional Young Investigator Award in 1988. He received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS Best Paper Award and two Best Paper Awards at the Design Automation Conference in 1983 and 1993. He is the Editor-in-Chief of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. He was the Program Chair (for Design Tools) of the Design Automation Conference in 1996 and 1997, and he is currently serving in the DAC Executive Committee. He was Program and General Chair of International Conference on Computer Design (ICCD) in 1988 and 1989, respectively.