Timed Supersetting and the Synthesis of Large Telescopic Units

L. Benini [#] G. De Micheli [#] A. Lioy [‡] E. Macii [‡] G. Odasso [‡] M. Poncino [‡]

[#] Stanford University
Computer Systems Laboratory
Stanford, CA 94305

[‡] Politecnico di Torino
Dip. di Automatica e Informatica
Torino, ITALY 10129

Abstract

In high-performance systems, variable-latency units are often employed to improve the average throughput when the worst-case delay exceeds the cycle time. Although such units have traditionally been hand-designed, recent results have shown that variable-latency units can be automatically generated. Unfortunately, the existing synthesis procedure has limited applicability due to its computational complexity.

In this work, we define and study an optimization problem, timed supersetting, *whose solution is at the kernel of the procedure for automatic generation of variable-latency units. We contribute a new algorithm for solving timed supersetting in the most difficult case, that is, when the timing behavior of the circuits is expressed through an accurate delay model. The proposed solution overcomes the complexity limitation of previous approaches, and its robustness is experimentally demonstrated by obtaining high-throughput, variable-latency implementations for all the largest circuits in the* Iscas'85 *and* Iscas'89 *benchmark suites.*

1 Introduction

As performance constraints become tighter, it is increasingly difficult to speed up combinational logic blocks simply by reducing their critical path delays. Variable-latency units (i.e., circuits that take a variable, integer number of clock cycles to complete a computation) are frequently used in high-throughput systems to achieve good average-case performance even when the worst-case delay can not be accommodated within the cycle time. Floating-point arithmetic units are typical examples of circuits of this kind.

The hand-crafted design of variable-latency units is a difficult task; for this reason, recently, a method for the automatic generation of such units has been proposed [1]. The key idea of this approach is that of transforming a slow, fixed-latency unit into a fast, variable-latency one (called *telescopic unit*) which delivers a higher average throughput with low average latency.

Consider a combinational *unit*, defined as the logic between two sets of latches. Given the minimum allowable cycle time, T, of the unit, equal to its longest delay, a reduced value, $T^* < T$, is selected. Then, the input conditions for which the propagation of the input values through the original logic takes longer than T^* are identified. Finally, a combinational block is automatically synthesized and added to the original unit. The task of such block is to generate a handshaking signal, the *hold signal* f_h, whose value informs the environment when the correct result is available at the outputs of the unit. Obviously, the hold logic introduces area, timing, and power overheads that must be kept under control. For this purpose, ad-hoc synthesis heuristics have been presented in [1].

The synthesis of telescopic units entails the solution of a general problem that we call *timed supersetting* (*TS* for brevity): *Find a set of input conditions that include all values propagating to the outputs with delay longer than a given* T^*.

The main theoretical contribution of this paper is to study the properties of TS, analyze its relationship with classical results in the field of timing analysis, and describe a novel class of algorithms for its solution.

From the practical side, the new algorithms overcome the main limitation of the methods of [1], namely the applicability of the synthesis procedures of the hold signal to large circuits, due to the inherent weakness of the exact, ADD-based algorithm employed [2]. When a complex and realistic delay model is adopted, the algorithm is memory and time consuming; therefore, it is usable only for small circuits, i.e., a few hundreds of gates. To partially alleviate this problem, one may resort to a simpler delay model, e.g., the unit delay model. Even in this case, however, the wall of a few thousands of gates can hardly be broken. This is not surprising, since the ADD-based method exactly solves the *false path* problem, which is known to be NP-complete [3], independently on the selected delay model. Our new algorithms replace the exact ADD-based method and enable the computation of the hold function for large blocks (several thousands of gates) even when a complex gate delay model is adopted.

The main advantage of the ADD-based technique is that it provides the true propagation delay for each input pattern; hence, it allows the computation of the *minimum set of patterns* that solves TS. In contrast, the algorithms proposed in this paper find a non-minimum solution to TS. Such solution is *conservative*, that is, it always includes the minimum one.

The downside of the conservative solution is that the hold logic may be activated for patterns that do not actually violate the cycle time constraint; thus, the telescopic unit may operate with an average throughput that is inferior to what could theoretically be achieved. Nevertheless, the advantages overcome the limitations, since the new algorithms for the solution of TS are practical for much larger and more complex units, such as those that can be found in high-performance microprocessors or DSPs.

Throughout the paper, the knowledgeable reader may observe the strong relationship between TS and the timing analysis problem (i.e., finding the true longest delay of a circuit and a pattern that exercises it). Indeed, a minimum solution to TS and the true longest delay of a circuit can be found by the same ADD-based algorithm; on the other hand, approximate timing analysis methods can not be directly used for solving TS.

The improved procedure for the automatic synthesis of telescopic units which encompasses the TS solution algorithms of this paper has been implemented within the SIS environment [4] using the CUDD package [5], and it has been benchmarked on the largest circuits taken from the Iscas'85 [6] and Iscas'89 [7] benchmark suites. Results are satisfactory, since an average throughput improvement of 12% has been achieved at the price of a 5.8% average area overhead.

2 Background

2.1 Boolean Functions and Operators

We assume the reader to be familiar with the basic concepts of Boolean functions and with the data structure commonly used for the symbolic manipulation of such functions, that is, the *binary decision diagrams* (BDDs). Background material on this subject can be found in [8].

We review here two Boolean operators which are essential for our purposes. Let $\mathrm{x} = [x_1, x_2, \ldots, x_n]$ be a vector of Boolean variables. Given a single-output Boolean function, $f(\mathrm{x})$, the *positive* and the *negative cofactors* of f, with respect to variable x_i, are defined as:

$$f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

and

$$f_{x'_i} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

The *existential abstraction* of f with respect to x_i is defined as:

$$\exists_{x_i} f(\mathrm{x}) = f_{x_i} + f_{x'_i}$$

The *Boolean difference* of f with respect to x_i is defined as:

$$\frac{\partial f(\mathrm{x})}{\partial x_i} = f_{x_i} \oplus f_{x'_i}$$

2.2 Circuits and Delays

A *combinational logic block* is a feedback-free network of combinational logic gates. 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 *non-controlling value* at a gate input is the value whose presence is not sufficient to determine the value at the output of the 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 logic block. 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 the primary input vector x is applied at time 0. Given a maximum delay constraint, the *required time*, $RT(g, \mathrm{x})$, is the time at which the output of gate g is required to be stable when the primary input vector x is applied in order for the output to stabilize within the maximum allowed delay. The *slack*, $ST(g, \mathrm{x})$, of a gate g is the difference between its required time and its arrival time, i.e., $ST(g, \mathrm{x}) = RT(g, \mathrm{x}) - AT(g, \mathrm{x})$.

A *path* in a combinational logic block is a sequence of gates and connections, $(g_0, c_0, \ldots, c_{n-1}, g_n)$, where connection c_i, $i = 0, 1, \ldots, n-1$, connects the output of gate g_i to the input of gate g_{i+1}. The *length* of a path, $\mathcal{P} = (g_0, c_0, \ldots, c_{n-1}, g_n)$ is defined as

$$d(\mathcal{P}, x) = \sum_{i=0}^{n-1} d(c_i, x)$$

The *topological delay* of a combinational logic block 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, $(e_0, e_1, \ldots, e_n)$, occurring at gates $(g_0, g_1, \ldots, g_n)$ 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 $\mathcal{P} = (g_0, c_0, \ldots, c_{n-1}, g_n)$ is said to be *sensitizable* if an event e_0 occurring at gate g_0 can propagate along $\mathcal{P}$. A *false path* is a non-sensitizable path. The *critical path* of a combinational logic block is the longest sensitizable path under a specified delay model: Its length is the delay, D, of the combinational logic block 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.

It is possible to compute *topological* approximations to arrival times ($AT(g)$), required times ($RT(g)$), slacks ($ST(g)$), and path lengths ($d(\mathcal{P})$) by resorting to known graph algorithms [9] whose complexity is linear in the number of gates involved. Such approximations have two important properties, namely they are conservative and pattern-independent. The first property is expressed by the following inequalities which hold for all x:

$$\begin{array}{lcl} AT(g) & \geq & AT(g, \mathrm{x}) \\ RT(g) & \leq & RT(g, \mathrm{x}) \\ ST(g) & \leq & ST(g, \mathrm{x}) \\ d(\mathcal{P}) & \geq & d(\mathcal{P}, \mathrm{x}) \end{array}$$

The second property is expressed by omitting the input dependence in the symbols representing the topological approximations. The topological delay of a combinational block, called *topological critical path*, is the topological delay of its longest path.

3 Telescopic Units

Suppose that the want to increase the average throughput of a combinational unit, shown in Figure 1-a. Obviously, this can be done by shortening the cycle time of the unit from its original value, T, to $T^* < T$. One possible way of achieving this goal is through the addition to the combinational unit of an output signal, f_h (called the *hold* output), which takes the value 1 anytime an input vector requires more than T^* time units to propagate to the outputs of the block (see Figure 1-b).

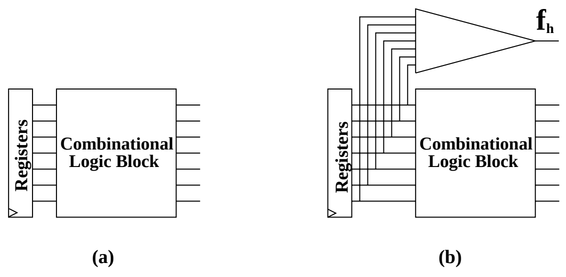

Figure 1: A Combinational Unit (a), and a Telescopic Unit (b).

The modified unit is called *telescopic unit*, since it may require additional cycles for terminating the computation, depending on the specific patterns appearing at the primary inputs of the unit. For patterns which are not activating the hold signal, the computation completes in $T^* < T$ time units. Patterns for which $f_h = 1$, on the other hand, require $2T^*$ time units to propagate through the logic block all the way to the primary outputs.

3.1 Conditions for Throughput Improvement

Clearly, the lower the probability of the hold signal to take on the value 1, the larger the overall throughput improvement achieved by the telescopic unit.

The average throughput, P^*, of the telescopic unit is given by the following formula:

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

where $Prob(f_h)$ is the probability of the *hold* signal to be one. On the other hand, the average throughput of the original unit is simply:

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

The use of the telescopic unit is therefore advantageous only for some values of T^* and $Prob(f_h)$, i.e., when $P^* > P$. We have the following condition for throughput improvement:

$$Prob(f_h) < \frac{2(T - T^*)}{T}$$

It should be noticed that the inequality above is valid only for $T^* \geq T/2$. Even though, in principle, the expression for P^* can be modified so as to account for values of $T^* < T/2$, it should be considered that in this case the circuitry needed to support the telescopic unit would become more complex, since the combinational logic may need, for some input patterns, more than two cycles to complete its computation. It is thus assumed that T^* is always $T/2 \leq T^* \leq T$.

3.2 Synthesis of the Hold Logic

The synthesis of the hold logic critically depends on the capability of finding all input patterns that propagate to the outputs with delay larger than T^*. Such patterns must be included in the ON-set of the hold function f_h. In the next section we will analyze this problem in detail. Here, we assume a black-box procedure, `ComputeF_h(`B`, `T^*`)`, which returns the ON-set of f_h (represented as a BDD) to be available. The input parameters of such procedure are the initial specification of the unit, B, and the desired cycle time, T^*.

`ComputeF_h` solves the timed supersetting problem. In fact, the minimum solution of TS is the ON-set of the hold function f_h that contains *all and only those* input values that propagate to the outputs of the unit with a delay longer than T^*. We would like to obtain solutions of TS that are as close as possible to the minimum one because their probability of being one is minimized, hence the average throughput is maximized.

However, it must be guaranteed that the implementation of the hold logic itself has a delay shorter than T^*, and this may not be always possible. Thus, the target is to determine an *enlarged hold function*, $f_h^e \geq f_h$, such that the average performance of the unit only marginally degrades, but the implementation of f_h^e meets the timing constraint, T^*, and has a limited area.

A heuristics has been devised in [1] for determining and synthesizing the enlarged function f_h^e; it starts from the BDD representation of f_h, and it 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.

4 The Timed Supersetting Problem

In this section we formally state the timed supersetting (TS) problem and one important variation, called *minimum timed supersetting* (MTS). The practical relevance of TS and MTS for the synthesis of telescopic units has been outlined in the previous section. For the sake of comparison, we briefly describe the algorithm for the solution of MTS (and TS) presented in [1]. We then take a completely different approach and present the key contribution of this paper, namely a robust and widely applicable algorithm for the solution of TS.

Consider a combinational logic block B with primary inputs $\mathbf{x} = [x_1, ..., x_{n_i}]$ and outputs $\mathbf{o} = [o_1, ..., o_{n_o}]$. The timed supersetting problem can be formally stated as follows:

Problem 1 *Find a set S of input values $\mathbf{x}$ that includes all values which propagate to the outputs $\mathbf{o}$ with a delay larger than, or equal to a given T^*.*

Obviously, TS has always the trivial solution B^{n_i}, i.e., the complete Boolean space is guaranteed to include all input values with propagation delay larger than T^*. We are interested in non-trivial solutions of TS. A theoretically relevant solution is the minimum one. The minimum timed supersetting problem consists of finding the smallest set of input values with propagation delay larger than T^*. Formally:

Problem 2 *Find the set S_{min} of all and only those input patterns $\mathbf{x}$ which propagate to the outputs $\mathbf{o}$ with a delay larger than, or equal to a given T^*.*

It is quite easy to prove the NP-completeness of MTS. Solving MTS when T^* is equal to the longest propagation delay of B is at least as hard as finding a single pattern with maximum propagation delay. This problem is NP-complete [3].

Observe that $S_{min} \subseteq S$, i.e., every solution of TS is guaranteed to contain the solution of MTS. Among the solutions of TS, we are interested in *near-minimum* solutions. More in detail, we are looking for approximations S of S_{min} that:

1. Include S_{min};
2. Are as close as possible to S_{min};
3. Can be computed in polynomial time and space (in n_i).

Before discussing our approximation strategy, we briefly review an algorithm for the exact solution of MTS.

4.1 Exact MTS Solution

An ADD-based algorithm for the exact solution of MTS has been presented in [1]. The arrival time ADD, $AT(g_{o_i}, \mathbf{x})$, for each output o_i of the block, is first computed using the algorithm of [2]. Such ADDs provide the propagation delay for any possible input vector. The BDD for the function $f_h^{o_i}$, which assumes the value 1 for all input vectors for which the arrival time of o_i is greater than the desired cycle time T^*, is then given by:

$$f_h^{o_i}(\mathbf{x}) = THRESHOLD(AT(g_{o_i}, \mathbf{x}), T^*)$$

THRESHOLD is the ADD operator that takes f, a generic ADD, and *val*, a threshold value, and sets to 0 all leaves of f whose value is smaller than *val* and to 1 all 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 [10].

Since the interest is in the set of input conditions for which *at least* one block output o_i has an arrival time greater than T^*, we have that f_h can be easily determined as:

$$f_h(\mathbf{x}) = \sum_{i=1}^{n_o} THRESHOLD(AT(g_{o_i}, \mathbf{x}), T^*)$$

where n_o is the total number of block outputs.
The main limitation of the algorithm is its worst-case exponential time and space complexity. When a complex, load and path dependent delay model is used, it is impossible to build the delay ADDs AT even for the outputs of relatively small circuits. The memory requirements for such construction are simply excessive. Another shortcoming of this approach is that when building the delay ADDs complete delay information is computed, even for patterns that propagate much faster than T^*. The computation of unneeded information contributes to the memory blow-up problem.

4.2 Near-Minimum TS Solution

Since the exact solution of MTS is computationally infeasible for large circuits, we resort to algorithms that solve TS but attempt to find solutions which are as close as possible to the minimum one. Notice the analogy with the approaches used in timing analysis. When exact delay computation is unaffordable, it is possible to resort to safe approximations with various degrees of tightness.
Consider a combinational logic block B with primary inputs $\mathbf{x} = [x_1, ..., x_{n_i}]$. A gate, g_i, of the network is associated with a Boolean function $f_i(\mathbf{y})$, where $\mathbf{y} = [y_1, ..., y_{n_{g_i}}]$ is the *local support* of f_i. We call $F_i(\mathbf{x})$, the Boolean function associated with gate g_i expressed as a function of the primary inputs (*global support*).
Let us assume that topological static delay analysis has been performed, and that the topological critical path $\mathcal{C} = (g_1, g_2, \ldots, g_m)$ has been determined (notice that, for the sake of clarity, we omit the indication of the connections between gates in the specification of paths). Let us assume also that the delay of the critical path, T_c, violates the desired cycle time, namely $T_c > T^*$. Since we are relying only on topological delay analysis, we conservatively consider the path as a true one. Consequently, all input conditions that activate it must be in the ON-set of the hold function f_h.
To find such conditions, from the primary inputs, we move along the critical path towards the output. We call *critical input* y_c of a gate g_i on the critical path the input which connects it with gate g_{i-1}, that is, connection c_{i-1}. For each gate in the path, we specify the *local sensitization function* s_i as the Boolean function that takes on the value 1 for all input conditions such that the critical input is a controlling input:

$$s_i(\mathbf{y}) = \frac{\partial f_i(\mathbf{y})}{\partial y_c} \quad (1)$$

Notice that, given $s_i(\mathbf{y})$ for a gate g_i, we can compute the *global sensitization function* S_i, which expresses the sensitization conditions for gate g_i as a function of the primary inputs $\mathbf{x}$. This can be done by recursive backward substitution of the local support variables until the primary inputs are reached.
The sensitization conditions for the entire path, $\mathcal{C}$, can now be computed as the intersection of all sensitization conditions of the gates $g_1, \ldots, g_m$ in $\mathcal{C}$. In formula:

$$S_{crit}(\mathbf{x}) = \prod_{i=1}^{m} S_i(\mathbf{x}) \quad (2)$$

We call S_{crit} the ***path sensitization conditions.*** This formula holds because for a signal to propagate along a path, all gates on the path must be sensitized. We call *partial path sensitization conditions* $S_{crit,j} = \prod_{i=1}^{j} S_i$ (with $j \leq m$) the path sensitization conditions for gates belonging to the path up to level j. Clearly, $S_{crit,m} = S_{crit}$. The partial sensitization conditions can be computed with the following recursion, for $j = 1, \ldots, m$:

$$\begin{array}{rcl} S_{crit,j}(\mathbf{x}) & = & S_{crit,(j-1)}(\mathbf{x}) \cdot S_j(\mathbf{x}) \\ S_{crit,1}(\mathbf{x}) & = & S_1(\mathbf{x}) \end{array} \quad (3)$$

An important property of Formula 3 is that $S_{crit,j} \leq S_{crit,k}$ for each $j > k$, that is, the $S_{crit,j}$'s are *monotonically* decreasing (i.e., the ON-set of S_{crit} is monotonically shrinking) with increasing j. Notice that computing the complete S_{crit} is equivalent to testing the viability of path $\mathcal{C}$. Since this is a NP-complete problem, there will be instances for which this computation requires an exponential amount of time or resources. However, the key observation is that we do not have to compute the complete S_{crit} to find a conservative set of input conditions for which the circuit delay T_c violates the timing constraint T^* (i.e., the hold function f_h). Any $S_{crit,j}$ is suitable for that purpose, because its ON-set contains the one of S_{crit}.
Although it may appear that Recursion 3 provides a viable procedure for finding a near-minimum solution of TS, two major problems need to be addressed:

1. It is a well-known fact that the absence of static path sensitization conditions (i.e., $S_{crit} = 0$) is *not* sufficient to guarantee that a path does not propagate events with delays that violate the timing constraint T^*. This phenomenon is known as dynamic sensitization [3]. Notice that every valid solution to TS must include all patterns with propagation longer than T^*. Hence, the approach based on simple static sensitization may lead to incorrect implementations and must be augmented by some form of dynamic sensitization test.
2. Recursion 3 has been obtained under the assumption that there is only one path violating the timing constraint. This is not generally true. In almost all practical examples there are multiple critical paths. Moreover, the number of such paths can be exponential in the number of gates in the network. The complexity explosion caused by the number of critical paths must be addressed in a conservative fashion.

In the next two sections we analyze and solve the above problems. We then describe in detail our strategy for finding a near-minimum TS solution in an efficient and robust way.

4.2.1 Accounting for Dynamic Sensitization

In order to derive sensitization conditions which are correct and conservative, let us consider the following situation at a critical gate g_i on a potentially critical path $\mathcal{P}$ (i.e., a path for which the topological delay estimate exceeds the time constraint T^*). Let $W = \{w_1, \ldots, w_p\}$ denote the set of side inputs, and $AT(w_i)$, $i = 1, \ldots, p$ their topological arrival times.
For a given input value $\mathbf{x}$, consider the propagation of signals on the critical path and on the side paths. Two situations may occur at gate g_i:

1. Every side input w_i arrives earlier than the critical input y_c, that is, $AT(w_i, \mathbf{x}) < AT(y_c, \mathbf{x}), \forall i$;
2. One or more side inputs arrive later than the critical input y_c, that is, $\exists i$ such that $AT(w_i, \mathbf{x}) > AT(y_c, \mathbf{x})$.

In the first case, we are safe, in the sense that when y_c arrives, all side inputs are already stable, and therefore the conditions of Equation 1 correctly represent the conditions under which the path $\mathcal{P}$ is sensitized.

In the second case, the situation is more uncertain. In fact, if y_c arrives earlier than any of the side inputs, it could propagate through gate g_i, and eventually get slowed down in such a way that it arrives late at the output. In this case, therefore, the conditions of Equation 1 may erroneously declare $\mathcal{P}$ as a false path. To solve this ambiguity, we have developed the following safe conditions for declaring a path as false.

Theorem 1

Given the topological arrival times, required times, and slacks for all gates belonging to path $\mathcal{P}$, the static sensitization conditions of Equation 1 are correct if, for all gates g_i of $\mathcal{P}$:

$$(AT(y_c) + ST(g_i)) > AT(w_i), \quad \forall w_i \in W \tag{4}$$

In essence, Theorem 1 states that, if the conditions of Inequality 4 hold, and $S_I(\mathbf{y}) = \frac{\partial f_i(\mathbf{y})}{\partial y_c} = 0$, then the path $\mathcal{P}$ is surely false. If Inequality 4 does not hold for some side inputs of a gate g_i, the static sensitization conditions of the gate can not be used for computing the path sensitization conditions.

This criterion may be extremely conservative in some cases, because it does not allow to use the sensitization conditions for a gate g_i if any of its side inputs does not satisfy Inequality 4. In the vast majority of cases, only some of the inputs violate the inequality. When the inputs that satisfy the inequality have controlling value, the gate still filters out events on the critical input. Therefore, we can relax the conditions stated by Equation 1. This can be done by exploiting some of the results available in the literature on timing analysis. A well-known criterion which is particularly suitable for a BDD-based symbolic implementation is the one introduced by Brand and Iyengar [11]. In that work, the sensitization conditions (Equation 1) are overestimated by *abstracting* a set of the local gate inputs. In formula:

$$\xi_i(\mathbf{y}) = \exists_{x_0,\ldots,x_k}\left(\frac{\partial f_i(\mathbf{y})}{\partial y_c}\right) \tag{5}$$

where $(x_0, \ldots, x_k)$ are some of the local inputs of gate g_i. When $k = 0$, ξ is exactly the static sensitization condition of Equation 1, whose robustness can thus be guaranteed. At the other extreme, when k equals the number of inputs of g_i, $\xi_i = 1$, regardless of the static sensitization conditions.

The key point with this approach is selecting which and how many inputs should be abstracted. Inequality 4 provides the criterion to do that. If we call $\widetilde{W} \subseteq W = (\widetilde{w}_0, \ldots, \widetilde{w}_k)$ the set of side inputs that do not satisfy Inequality 4, the comprehensive criterion for robustly and correctly detecting a false path, at a gate g_i, becomes:

$$\sigma_i(\mathbf{y}) = \exists_{\widetilde{w}_0,\ldots,\widetilde{w}_k}\left(\frac{\partial f_i(\mathbf{y})}{\partial y_c}\right) \tag{6}$$

Similarly to the static sensitization conditions, we can extend $\sigma_i(\mathbf{y})$ to the global support of the combinational block, and compute $\Sigma_i(\mathbf{x})$ as a function of the primary inputs $\mathbf{x}$. The sensitization conditions for the entire path are then given by the intersection of all sensitization conditions of the gates $g_1, \ldots, g_m$. In formula:

$$\Sigma_{crit}(\mathbf{x}) = \prod_{i=1}^{m} \Sigma_i(\mathbf{x}) \tag{7}$$

In summary, the correct and conservative sensitization conditions $\Sigma_{crit}(\mathbf{x})$ for a single path $\mathcal{P}$ are obtained through the procedure summarized in Figure 2.

```
Compute_Sigma ($\mathcal{P}$) {
  $\Sigma_{crit}(\mathbf{x}) = \mathbf{1}$;
  for (each gate $g_i \in \mathcal{P}$) {
    /* $y_c$ = critical input; /*
    /* $AT$ = arrival times; $ST$ = slacks */
    $s_i(\mathbf{y}) = \frac{\partial f_i(\mathbf{y})}{\partial y_c}$;
    for (each side input $w_i$ of $g_i$)
      if (($AT(y_c) + ST(g_i)) \leq AT(w_i)$)
        $s_i(\mathbf{y}) = \exists_{w_i} s_i(\mathbf{y})$;
    $\sigma_i(\mathbf{y}) = s_i(\mathbf{y})$;
    $\Sigma_i(\mathbf{x}) = \mathrm{Extend}(\sigma_i(\mathbf{y}), \mathbf{x})$;
    $\Sigma_{crit}(\mathbf{x}) = \Sigma_{crit}(\mathbf{x}) \cdot \Sigma_i(\mathbf{x})$;
  }
}
```

Figure 2: Algorithm for Computing $\Sigma_{crit}(\mathbf{x})$.

Procedure Extend expresses the sensitization conditions in the global support $\mathbf{x}$. Inequality 4 is used to decide which side inputs arrive too late and should be quantified out from the sensitization conditions. It is important to notice that the procedure does not solve MTS exactly, because conservative and pattern-independent topological estimates of the arrival times and slacks are used. In other words, Inequality 4 is a sufficient but not necessary condition for deciding whether a side input stabilizes before the arrival of the critical input.

4.2.2 Dealing With Multiple Paths

So far, we have described a robust, yet simple algorithm for finding a near-minimum TS solution which is applicable to the cases where a single critical path is present in the circuit. In this section, we present an algorithm (see the pseudo-code in Figure 3) to find a near-minimal solution to TS (i.e, the hold function f_h for a telescopic unit) in the case of multiple critical paths.

```
  procedure ComputeF_h($B$,$T^*$) {
1   StaticTimingAnalysis($B$);
2   $Levels$[] = Levelize($B$);
3   foreach (level $l = 0, \ldots, L$) {
4     foreach (critical gate $n \in Level[l]$) {
5       $PAF_n(\mathbf{x}) = \mathbf{0}$;
6       foreach (critical fanin $i$ of gate $n$) {
7         $PAF_n(\mathbf{x}) = PAF_n(\mathbf{x}) + (PAF_i(\mathbf{x}) \cdot \Sigma_i(\mathbf{x}))$ ;
        }
      }
    }
    $f_h(\mathbf{x}) = \mathbf{0}$;
8   foreach (critical gate $n \in Output\_gates$) {
9     $f_h(\mathbf{x}) = f_h(\mathbf{x}) + PAF_n(\mathbf{x})$;
    }
  }
```

Figure 3: Algorithm for Computing the Hold Function.

The procedure ComputeF_h receives, as inputs, the combinational block description, B, and the desired cycle time, T^*, either as an absolute time value, or as a percentage of the actual critical delay of the circuit, in the case this is not known a priori.

It initially performs (Line 1) a static timing analysis of the circuit, computing arrival times, required times, and slacks for each gate. Then, the network is levelized (Line 2), that is, the gates are grouped into the list *Levels*[] according to their topological level, starting from the primary inputs, which are assumed to be at level 0. Starting from level 0, the critical gates (i.e., gates with negative slack) at each level are processed (Line 4), and a local Boolean function $PAF(\mathbf{x})$ (*Path Activation Function*) is computed as follows: At each gate, the function is obtained by summing, over its critical fanins, the product of two quantities: The path activation function of the i-th fanin ($PAF_i(\mathbf{x})$ in Line 7), and the sensitization conditions $\Sigma_i(\mathbf{x})$, computed with Equation 7. Clearly, the path activation function for each primary input is assumed to be 1. The output of the procedure is a near-minimum solution of TS, or equivalently, the hold function f_h of a telescopic unit. It is computed in Lines 8 and 9, by accumulating the PAFs of all critical gates that are connected to an output.

The rationale of the algorithm is that every critical gate filters the activation conditions of a critical input i by AND-ing the Σ_i of the input to the conditions for which an event propagates up to input i (i.e., PAF_i). If a gate has more than one critical input, its PAF is the sum of the filtered PAFs of its critical inputs.

An important feature of the algorithm is that it is based on a traversal of the critical gates, and not of the critical paths. In fact, the number of (critical) paths can be exponential in the number of gates in the network, whereas the number of critical gates is guaranteed to be smaller than the number of gates.

Note that the algorithm relies on topologic delay estimates. It is a well-known fact that such estimates can be very conservative. In the limit case, if the topologic delay is longer than the true delay, and the true delay is shorter than T^*, we may actually synthesize useless hold logic and make the throughput worse. This is due to the fact that our procedure is conservative and it may actually flag as belonging to f_h some input conditions that do not propagate any perturbation to the output. Observe, however, that the accuracy of the procedure can be improved if more powerful algorithms for the computation of the arrival times are used (see, for example, [12, 13]). The modification of the pseudo-code in Figure 3 is straight-forward: It is sufficient to replace the `StaticTimingAnalysis` call with the call to an advanced timing analysis procedure. On the other hand, the computational burden of obtaining accurate delay information for all gates in the network may be substantially higher than that required by simple static timing analysis. In summary, the `StaticTimingAnalysis` should be replaced by the procedure that is used for timing analysis in the design flow.

4.2.3 Cutting Heuristics

Although the algorithm of Figure 3 does not suffer the computational bottleneck of the exact method of [1], there may be circuits for which constructing the BDDs for the sensitization function is still not feasible. In these cases, an approximate solution is required, that allows to compute *partial* timing information.

A simple solution may be that of stopping procedure `ComputeF_h` after a desired number of levels or, alternatively, when the sizes of the BDDs grow beyond a given threshold. Unfortunately, this would result in incomplete timing information, since some critical paths could be incorrectly left out of the computation. In fact, computing the hold function by levels does not necessarily take into account all critical paths, unless we guarantee that last level (i.e., the primary outputs) is reached.

The observation above suggests a criterion for computing the timing information *incrementally*. Intuitively, the key for such criterion should be to progressively select sets of critical gates, hereafter called *cuts*, such that the gates in a set cut all critical paths. If we can compute the BDDs (in the global support) of the path activation functions of all gates in a cut G, a solution of TS (i.e., a valid f_h) is simply:

$$f_h = \sum_{n \in G} PAF_n(\mathbf{x}) \tag{8}$$

A good cutting heuristics is obviously essential for an effective realization of the f_h computation algorithm. The one we propose starts from the critical inputs (cut G_0), and consists of the repeated application of three phases, until no gate in the combinational block is left.:

1. From a cut G_i, we reach the critical gates in the fanout of any gate in G_i. Only critical connections (i.e., connections from the output of a critical gate to the critical input of another critical gate) are explored. A newly reached gate is marked as belonging to the new cut G_{i+1} only if all its critical fanins belong to a previous cut G_j, $j = 0, 1, ..., i$, or to cut G_{i+1} itself.
2. If at least one gate has been marked, we check if all critical fanins of some additional gates reached from G_i have been reached. If this is the case, such gates are marked as belonging to G_{i+1}. This step is repeated until no new gate is marked. In other words, all gates for which all the critical fanins belong to G_j, $j \leq i + 1$ are marked.
3. The remaining critical gates reachable from G_i do not belong to G_{i+1} and are *discarded*. However, to guarantee that all critical paths are cut, we insert in G_{i+1} all critical fanins of the discarded nodes which belong to previous cuts (or to cut G_{i+1} itself).

The set of gates G_{i+1} is the new cut. Notice that, if an output is reached during traversal at cut G_j, such output is inserted in all successive cuts G_k, $k > j$. In addition, it can be easily observed that, in general, cuts are not disjoint.

After the computation of G_{i+1}, the path activation functions of its gates and f_h are computed. The termination conditions of the traversal algorithm are the following:

- If the BDD of a PAF for a gate in G_{i+1} blows up, the computation is aborted and the BDD of the f_h of the previous cut is returned.
- Once all PAFs have been computed, f_h is obtained by taking the Boolean sum of all PAFs. If the BDD of f_h blows up during the Boolean sum, the computation is aborted and the BDD of the f_h of the previous cut is returned.
- If the computation of f_h in the global support succeeds, and the cut is the last one, f_h is returned. Conversely, if the cut is not the last one, f_h is stored and the next cut is generated.

The f_h for G_0 is obviously the most conservative TS solution, that is, $f_h = 1$. In the worst case, if PAF or f_h computation fails at the first cut, the value of f_h returned is the tautology. Hence, the procedure is guaranteed to return a valid solution to TS, but it may return the trivial one.

The algorithm for near-minimum TS solution described in Section 4.2.2 is modified by replacing a level-based traversal with a cut-based traversal. In this way, `ComputeF_h` is guaranteed to always return a valid solution to TS, even in the case of BDD blow-up.

Circuit	I	O	G	T	P	Prob(f_h)	G*	T^*	P^*	$T(f_h)$	ΔP	ΔG	Time
c1908	33	25	1339	23.51	0.04253	0.05761	1410	20.08	0.04836	17.46	13.7%	5.2%	63.2
c2670	233	140	1617	52.43	0.01907	0.15902	1723	44.50	0.02068	41.30	8.4%	6.5%	29.2
c3540	50	22	1876	29.28	0.03414	0.08666	2007	25.35	0.03773	24.94	10.5%	6.9%	402.9
c5315	178	123	3286	29.49	0.03391	0.24112	3443	24.06	0.03655	23.61	7.8%	5.3%	45.0
c7552	207	108	5439	23.16	0.04317	0.37500	5817	16.84	0.04824	16.52	11.8%	6.9%	2329.0
s3271	142	130	1393	41.63	0.02402	0.09180	1485	36.54	0.02611	35.85	8.7%	6.6%	6.0
s3330	172	205	1434	16.36	0.06112	0.01721	1511	14.56	0.06809	14.42	11.4%	5.3%	89.7
s3384	226	209	2307	32.91	0.03038	0.01294	2474	30.15	0.03295	29.14	8.5%	7.2%	166.0
s4863	153	120	3684	34.83	0.02871	0.06831	3828	27.63	0.03495	27.06	21.7%	3.8%	472.4
s5378	199	213	2078	36.80	0.02771	0.07885	2229	30.60	0.03139	28.89	13.2%	7.2%	138.5
s6669	322	294	4975	52.54	0.01903	0.04002	5210	45.93	0.02133	42.98	12.1%	4.7%	302.0
s9234	247	250	2821	19.32	0.05175	0.10112	3119	15.43	0.06153	15.34	18.9%	10.5%	2248.3
s13207	700	790	3554	22.61	0.04422	0.38551	3968	15.83	0.05099	15.05	15.3%	11.6%	76.7
s15850	611	684	4850	165.25	0.00605	0.08029	5377	132.20	0.00726	91.70	20.1%	10.8%	219.2
s35932	1763	2048	12944	874.98	0.00114	0.00001	13367	838.10	0.00119	133.11	4.4%	3.1%	15.6
s38417*	1664	1742	10326	60.31	0.01658	0.42663	10615	44.91	0.01751	39.28	5.6%	2.8%	1317.1
Average											12.0%	5.8%	

Table 1: Experimental Results.

5 Experimental Results

We have implemented the algorithms for TS solution described in Section 4 within the tool for telescopic units synthesis of [1]. The underlying logic synthesis system is SIS [4] which uses CUDD [5] as BDD package. Experiments have been run on a DEC AXP 1000/400 with 256 MB of memory.

We have considered *ALL* the circuits in the Iscas'85 [6] suite with more than 1000 gates. Since only 6 examples were available, we have also experimented with the combinational logic of the 12 largest Iscas'89 [7] (addendum included) benchmarks.

The library used for mapping consisted of 2 to 4-input NAND and NOR gates, plus inverters and buffers, each of which had five different driving strengths. The gates are non-symmetric, that is, they have different pin-to-pin delays, as well as different rise and fall delays. The delay model used is the SIS *real delay*. The original circuits have been first optimized for speed using a modified version of `script.delay`, where the `full_simplify`, and sometimes the `rr` commands have been removed to allow the optimization to complete on the large examples, and then mapped for speed with either `map -m1` or `map -n1 -AFG`.

Table 1 reports the experimental data. Columns *Circuit*, *I*, *O*, *G*, T and P give the name, number of inputs, outputs, and gates, the static delay (in *nsec*), and the throughput of the original circuit. Column *Prob*(f_h) shows the probability of f_h, column G^* gives the total number of gates of the telescopic unit, column T^* reports the cycle time (in *nsec*) at which the telescopic unit is clocked to achieve the increased throughput of column P^*, and column $T(f_h)$ tells the (static) arrival time of the hold signal (in *nsec*). Columns ΔP and ΔG give the throughput improvement and the area overhead (in terms of gates) of the telescopic unit. Finally, column *Time* reports the CPU time (in *sec*) required to perform the automatic synthesis of f_h for a given T^*. A symbol $*$ beside the circuit name indicates that the heuristics of Section 4.2.3 was required to complete the calculation of f_h. This has happened only on example s38417, where the computation of f_h stopped after 19 cuts (out of 28).

Only two benchmarks are missing from the table: c6288 and s38584. The former is a 32-bit multiplier, for which it is well known that the computation of the BDDs for all outputs is infeasible [14]. The application of the algorithm of Figure 3 therefore failed; we thus resorted to the heuristic method discussed in Section 4.2.3; also in this case, however, the result was negative, since the computation of f_h stopped after 41 cuts (out of 103) with $T^* = 0.95T = 70.40$ *nsec* and *Prob*(f_h) $= 0.99999$. The application of our algorithm to the latter example, on the other hand, has not been tried since a mapped version of it could not be obtained for the selected gate library.

The results are quite satisfactory, since an average throughput improvement of 12% has been achieved, with an average area penalty of 5.8%. The proposed approach thus demonstrates its scalability and applicability to the largest available benchmarks. Needless to say, all circuits examined here are well beyond the capability of the exact MTS solution algorithm of [1], which, for this library and delay model, fails for circuits larger than a few hundreds of gates. On small circuits, for which exact MTS solution is possible, our tool still achieves improvements around 10-15%, while the exact minimum solution allows average improvements around 26% [1].

6 Conclusions

We have addressed the timed supersetting problem, and we have contributed an algorithm for its solution which is well suited for the automatic synthesis of telescopic units. Results obtained on the largest benchmarks available in the literature are satisfactory, and confirm that the use of telescopic units represents a robust and flexible alternative for improving the performance of delay-critical digital applications.

References

[1] L. Benini, E. Macii, M. Poncino, "Telescopic Units: Increasing the Average Throughput of Pipelined Designs by Adaptive Latency Control," DAC-34, pp. 22-27, 1997.

[2] R. I. Bahar, et al., "Timing Analysis of Combinational Circuits using ADDs," EDTC-94, pp. 625-629, 1994.

[3] P. C. McGeer, R. K. Brayton, Integrating Functional and Temporal Domains in Logic Synthesis, Kluwer Academic Publishers, 1991.

[4] E. M. Sentovich, et al., "Sequential Circuits Design Using Synthesis and Optimization," ICCD-92, pp. 328-333, 1992.

[5] F. Somenzi, CUDD: University of Colorado Decision Diagram Package, Release 2.1.2, Tech. Report, Univ. of Colorado, 1997.

[6] F. Brglez, H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," ISCAS-85, pp. 785-794, 1985.

[7] F. Brglez, D. Bryan, K. Koźmiński, "Combinational Profiles of Sequential Benchmark Circuits," ISCAS-89, pp. 1929-1934, 1989.

[8] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Trans. on Computers, Vol. 35, pp. 79-85, 1986.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, An Introduction to Algorithms. McGraw-Hill, 1990.

[10] R. I. Bahar, et al., "Algebraic Decision Diagrams and their Applications," Formal Methods in System Design, Vol. 10, pp. 171-206, 1997.

[11] D. Brand, V. S. Iyengar, "Timing Analysis Using Functional Analysis," ICCAD-86, pp. 126-129, 1986.

[12] H. C. Chen, D. H. C. Du, "Path Sensitization in Critical Paths," IEEE Trans. on CAD, Vol. 12, pp. 196-207, 1993.

[13] S. Devadas, K. Keutzer, S. Malik, "Computation of Floating Mode Delay in Combinational Circuits: Theory and Algorithms," IEEE Trans. on CAD, Vol. 12, pp. 1913-1923, 1993.

[14] R. E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication," IEEE Trans. on Computers, Vol. 40, pp. 205-213, 1991.