Timing Optimization by Restructuring Long Combinatorial Paths

Jürgen Werber
Research Institute for Discrete Mathematics, University of Bonn
Lennéstraße 2,
53111 Bonn, Germany

Dieter Rautenbach
Department for Mathematics and Science,
Technical University of Ilmenau
Weimarer Straße 26,
98693 Ilmenau, Germany
Dieter.Rautenbach@tu-ilmenau.de

Christian Szegedy
Cadence Berkeley Labs
1996 University Ave,
Berkeley, CA 94704, USA
szegedy@cadence.com

***Abstract*—We present an implementation of an algorithm for constructing provably fast circuits for a class of Boolean functions with input signals that have individual starting times. We show how to adapt this algorithm to logic optimization for timing correction at late stages of VLSI physical design and report experimental results on recent industrial chips. By restructuring long critical paths, our code achieves worst-slack improvements of up to several hundred picoseconds on top of traditional timing optimization techniques.**

I. Introduction

Towards the end of the VLSI design process, physical design determines locations for all the gates in a netlist with the objectives of meeting timing constraints and minimizing interconnect wirelength, among others. Traditional timing optimization techniques include placement modifications, gate-sizing by choosing different standard cells offered by the technology library, V_t level assignment, interconnect buffering (inserting or reimplementing repeater trees), wire type assignment, and localized logic changes such as pin swapping or simple logic replacements that involve a very small number of gates.

Decisions taken during initial logic synthesis may have produced paths with a high number of non-repeater gates. At that early design stage, lack of physical information and uncertainty of estimates make it difficult to predict timing problems accurately. When a path later turns out to be timing-critical due to the number of individual gates that contribute to the total path delay, standard transforms such as those mentioned above cannot get to the root of the problem because they do not fundamentally change the logical structure of the netlist. On the other hand, the production time frame often forbids going back and restarting from logic synthesis. In this situation, tools are needed that avoid repeating previous design steps and optimize critical paths of high combinatorial depth after physical design has revealed them as critical in the context of a placed and timing-optimized netlist.

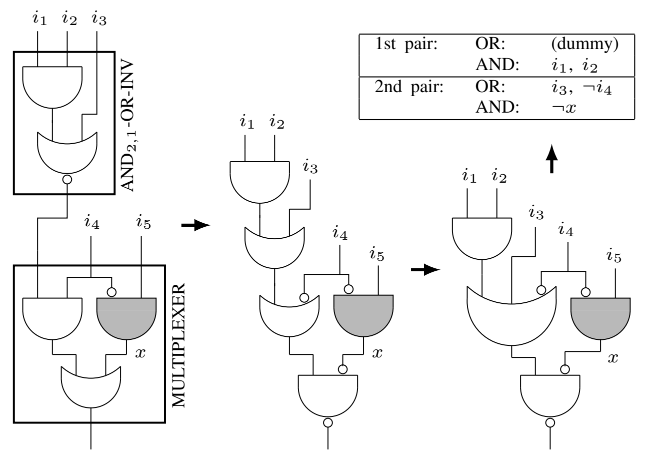
We present BONNLOGIC, a new tool that addresses this need. It is part of the BONNTOOLS [3], a comprehensive software package for physical VLSI design. BONNLOGIC restructures long critical paths and is guided by timing information that has become available during physical design. It is based on an algorithm [6] that constructs provably fast circuits with respect to a measure called circuit delay, which is a generalization of the classical notion of circuit depth and represents the stabilization time of the function outputs when inputs do not arrive simultaneously, as is typically the case in the VLSI application. While a simple delay model has been suitable for the mathematical analysis in [6], we show that the algorithm can easily be adapted to use physical information such as placement locations and data from static timing analysis, with a delay model that includes gate as well as interconnect delay. We demonstrate that, in addition to its theoretical justification, the method is also successful in practice, and present experimental results on industrial designs of up to six million gates where the overall worst slack is improved by up to several hundred picoseconds.

Logic replacements for timing optimization in physical design are often done by local exchange operations [12][13]. In contrast to these techniques, our approach explores more extensive replacements as it restructures entire paths. The importance of arrival time differences for the design of logic circuits has been recognized in a couple of recent publications: in [4] and [15], adder circuits are constructed by dynamic programming or greedy algorithms that take unequal arrival time profiles into account, but no general bounds have been proved (optimality claims for these circuits are restricted to the class of circuits that the algorithm can find). Earlier results on the design of specialized circuits taking input arrival times into account can be found in [5][9][10]. The essential difference and strength of our approach is that it works for all arrival time profiles and general paths and has a provable performance guarantee in a reasonable mathematical delay model.

Our paper is organized as follows. In Section II, we shall describe how to identify problem instances for our algorithm. Section III presents the core algorithm and gives a short summary of the results of [6]. In Section IV, we show how to adapt this algorithm to the VLSI world. Section V gives experimental results.

II. Finding Problem Instances on the Critical Path

During physical design, static timing analysis identifies signal paths through the logic that are too slow for the chip to operate correctly. The difference between required and



1st pair:	OR:	(dummy)
	AND:	i_1, i_2
2nd pair:	OR:	i_3, $\neg i_4$
	AND:	$\neg x$

Fig. 1. Processing a resolvable subpath. The subpath consists of two gates (an AND-OR-INV and a multiplexer). In this example, the leftmost input pin and the output pin are interlinked on both gates. (If the input pin corresponding to signal i_4 were the interlinked input pin instead of its neighbour, the path to the output pin would not be unique, so the multiplexer gate would not be resolvable in that case.) Main-path gates are drawn blank, while the side-logic gate is grey. The two other circuits are the results of the normalization steps explained in Subsection IV-A: in the middle circuit, inner inversions on the main path have been removed; in the circuit on the right, the two consecutive OR gates have been contracted such that the main path is an alternating chain of AND and OR gates. The table at the top lists the main-path inputs, grouped in pairs of OR and AND sets. A dummy OR set had to be added for the first pair. Note that i_5 is not a main-path input because it does not feed the main path directly, while i_1 is a main-path input in spite of being interlinked because it feeds the first main-path gate.

computed arrival times is called **slack**. In particular, negative slack indicates that timing constraints are not met. A worst-slack path is called a **critical path**, and the gates on such a path as well as the pins on the path can be queried from the static timing engine. We call these pins **interlinked**.

On a critical path, we shall distinguish between resolvable and non-resolvable gates. A gate on the critical path with interlinked input pin p and interlinked output pin q is **resolvable** if the following condition is satisfied: in the template circuit provided by the technology library which specifies the Boolean function(s) evaluated by the gate, the intersection of the fanout cone of p and the fanin cone of q is a path consisting only of AND or OR gates with arbitrary fanin and arbitrary pin inversions (including repeaters). Our algorithm deals with subpaths on which all gates are resolvable.

By gluing the template circuits together, we get a representation of the function computed by an entire resolvable subpath. It is a circuit made up of a **main path**, which contains the interlinked pins, plus **side logic**. An example is shown on the left in Fig. 1. We shall not modify the side logic gates in our core algorithm.

III. Restructuring Alternating Paths

In this section, we give the mathematical description and analysis of the core routine of our algorithm. Let us consider the Boolean function $f : \bigcup_{n\geq 1}\{0,1\}^{2n} \to \{0,1\}$ on all Boolean vectors of even dimension, defined by

$$\begin{aligned} &f(x_1, y_1, x_2, y_2, \ldots, x_n, y_n) \\ &\quad := \quad ((\ldots(((x_1 \wedge y_1) \vee x_2) \wedge y_2) \vee \ldots) \vee x_n) \wedge y_n. \end{aligned} \tag{1}$$

As shown in [6], the key to our algorithm is the observation that we can rewrite f as

$$\begin{aligned} &f(x_1, y_1, \ldots, x_n, y_n) \\ &\quad = \quad \big[f(x_1, y_1, \ldots, x_l, y_l) \wedge p(y_{l+1}, \ldots, y_n)\big] \\ &\quad\quad\quad \vee f(x_{l+1}, y_{l+1}, \ldots, x_n, y_n) \end{aligned} \tag{2}$$

for any $l \in \{1, \ldots, n-1\}$, where $p : \bigcup_{k\geq 1}\{0,1\}^k \to \{0,1\}$ denotes the conjunction $p(y_1, \ldots, y_k) := y_1 \wedge \ldots \wedge y_k$ of arbitrarily many variables. The following recursive formulation of the algorithm generates a circuit that has two output functions, f and p. Although we are really interested in the f part of the circuit, p is needed on lower recursion levels.

Input:	An integer $n \geq 1$; input arrival times $t(x_1), t(y_1), \ldots, t(x_n), t(y_n)$.
Output:	A circuit that computes $f(x_1, y_1, \ldots, x_n, y_n)$ and $p(y_1, \ldots, y_n)$.

```
If $n = 1$:
  • Return a circuit with one AND2 gate connected to inputs $x_1$ and $y_1$, where $f$ is computed by the gate and $p$ is computed by input $y_1$.
If $n > 1$:
  • For $l := 1$ to $n - 1$:
      – Use recursive calls to construct circuits for the two instances $(l;\ t(x_1), t(y_1), \ldots, t(x_l), t(y_l))$ and $(n - l;\ t(x_{l+1}), t(y_{l+1}), \ldots, t(x_n), t(y_n))$.
        Let $(f_{1,l}, p_{1,l})$ and $(f_{l+1,n}, p_{l+1,n})$ denote the respective functions.
      – Combine the two circuits and add two AND2 gates and one OR2 gate to form a new circuit computing $(f_{1,l} \wedge p_{l+1,n}) \vee f_{l+1,n}$ and $p_{1,l} \wedge p_{l+1,n}$.
  • Among these $n - 1$ new circuits, return one which minimizes the arrival time of the output.
```

This algorithm can be implemented in cubic time using dynamic programming (see Subsection IV-C).

In the above description, we have left it open how arrival times are defined. Indeed, the algorithm does not depend on a particular delay model. In the mathematical analysis, we use a simplified model; in the implemention, we take technology and physical information into account.

For the mathematical analysis, we assume to be given arrival times $t(z_1), \ldots, t(z_n) \in \{0, 1, 2, \ldots\}$ for the primary inputs $z_1, \ldots, z_n$ of a circuit C and recursively define the arrival time $t(g)$ of a gate g to be one plus the maximum of the arrival times of all predecessors of g. Following [6], we call $\text{delay}(C) := \max_v t(v)$ the **delay** of a circuit C, where the maximum is taken over all inputs and gates of C. (For the case where all input arrival times are zero, this definition of delay coincides with the classical notion of circuit depth [8].) For an arbitrary Boolean function F, arrival times for its inputs, and a basis Ω that specifies which functions may be used as gates, let $\text{delay}_\Omega(F)$ be the minimum delay of any circuit over Ω that computes F.

In order to state an approximation guarantee for the circuits we are going to construct, we need the following lower bound on delay:

Lemma 1: Let Ω be a basis that contains only functions on one or two variables. If a Boolean function F essentially depends on n inputs $z_1, \dots, z_n$ with arrival times $t(z_1), \dots, t(z_n)$, then

$$\mathrm{delay}_{\Omega}(F) \geq \left\lceil \log_2 \left(\sum_{i=1}^{n} 2^{t(z_i)} \right) \right\rceil .$$

Proof: Let C be a circuit over Ω that computes F. We can construct a binary tree of depth at most $\mathrm{delay}(C)$ by deleting edges from the graph underlying C until each vertex has at most one outgoing edge and replacing each input z_i by a full binary tree of depth $t(z_i)$ rooted at z_i. This tree has $\sum_{i=1}^{n} 2^{t(z_i)}$ leaves, so its depth is at least $\left\lceil \log_2 \left(\sum_{i=1}^{n} 2^{t(z_i)} \right) \right\rceil$. □

We now state the main result of [6], the proof of which is beyond the scope of this paper:

Theorem 1: Let f be as in (1), and let C be the circuit over $\Omega := \{\text{AND2, OR2, INV}\}$ generated by the algorithm for input arrival times $t(x_1), \dots, t(y_n)$. Then

$$\begin{aligned} \mathrm{delay}(C) &\leq 1.441 \cdot \log_2 \left(\sum_{i=1}^{n} \left(2^{t(x_i)} + 2^{t(y_i)} \right) \right) + 3 \\ &\leq 1.441 \cdot \mathrm{delay}_{\Omega}(f) + 3. \end{aligned}$$

Moreover, C uses at most $4n - 3$ gates, and the maximum fanout is 3 for inputs and 2 for gates. □

The factor 1.441 of the theorem is not the smallest possible. As shown in [7], it can be reduced to $(1 + \epsilon)$ for arbitrarily small $\epsilon > 0$, albeit with an additive constant growing polynomially in ϵ^{-1}. However, the algorithm of Theorem 1 is useful in practice because it yields circuits of small delay and linear size with bounded fanout, and all constants that appear are small. Moreover, it is easy to replace the notion of circuit delay by other objective functions that model arrival times according to the application. We shall now show how to do this in the context of timing optimization during physical design.

IV. Implementation of the Method

A. Preparing a Resolvable Subpath for the Dynamic Program

In order to run the dynamic programming algorithm for the function f defined in (1), we apply two normalization steps to the main-path logic. First, we get rid of "inner" inversions on the main path by applying De Morgan's rule on its gates, cancelling inversion pairs where appropriate (Fig. 1, middle circuit). After this, inversions can only appear at pins fed by primary subpath inputs or side logic, at the output of the last main-path gate, and on side logic gates. (Note that primary inputs in the subpath circuit may or may not be primary inputs on the whole chip.) Running De Morgan's rule over the whole circuit gives an alternative instance for the dynamic program; we shall use this fact in Subsection IV-F.

As a second step, we contract each consecutive sequence of AND gates to a single AND gate and each consecutive sequence of OR gates to a single OR gate (merging repeaters with either type as appropriate). Now AND and OR gates alternate on the main path (Fig. 1, bottom right circuit). Except for the very first gate on the path, the first input of each AND gate is fed by an OR gate without in-between pin inversion, and the first input of each OR gate is fed by an AND gate without in-between inversion. The rest of the input pins on the main path gates are fed by primary subpath inputs or gate output pins, and we call these **main-path inputs**.

When resolving the original subpath gates, the algorithm ensures that side logic gates cannot be further merged into the main path. For example, with no inversion between them, a side logic AND will not drive a main-path AND, but a main-path OR.

We group the main-path inputs in sets labelled alternately with AND or OR, according to the function of the gate they feed. The sets correspond to the input variables of (1). Combining the members of a set into a single signal is a subproblem that we shall deal with in Subsection IV-B, after explaining the delay model we use and how it guides the construction of new logic.

In order to obtain an alternating OR-AND path with gate types that match the definition of f, we might have to add a dummy OR set at the beginning of the path, or a dummy AND set at the end, or both. For example, if x_1 is missing, a dummy OR set at the beginning transforms $(y_1 \vee x_2) \wedge y_2$ into $((1 \wedge y_1) \vee x_2) \wedge y_2$. Dummy sets get a special treatment in our implementation (Subsection IV-C). The main-path input sets can now be grouped into pairs, where each pair consists of an OR set followed by an AND set (Fig. 1, top right).

The resulting sets can be degenerate: if there is only one main-path input, no AND or OR function is involved at all. This is an instance for a repeater tree algorithm. If there is just one AND set or one OR set, but no alternation between the two, it suffices to solve the above-mentioned subproblem without running the algorithm for f.

B. The Delay Model for the Dynamic Program

Our goal will be to map the solution circuit of the dynamic program into the netlist using only NAND2 and NOR2 gates since these are typically faster than non-inverting gates in CMOS technologies. (It is just for ease and readability that we have been using AND and OR gates in the algorithm description and analysis.) By De Morgan's rule, we can switch between AND and OR by flipping all pin inversions. Until the end of the dynamic program, we use AND2 and OR2 gates with arbitrary pin inversions, and these may or may not be switched to the respective other type later. The final decisions between NAND2 and NOR2 are postponed because they cannot be made independently for the individual gates without creating inverters for Boolean correctness, which we want to avoid because of their additional gate delay.

The dynamic program constructs its solution circuit starting at the inputs and working towards the complete function: when a partial solution is combined with others to give a new function in the course of the algorithm, this new signal will in turn be combined with other partial (f, p) pairs. It can play either role of the two subfunctions in (2), a distinction which produces even or odd numbers of inversions on the way to the larger partial solution. This is why fixing gates to NAND2 and NOR2 would eventually lead to additional inverters. Instead, we use a De Morgan sweep in the end when the structure of the entire function has been determined.

In our delay model, there is one parameter d_{gate} for the delay through a gate of fanin 2 (NAND2 or NOR2 in the final netlist). Although NAND2 is faster than NOR2 in general [11], our experience shows that the difference between the two types does not dominate the delay differences seen between different gates of the same type in logic optimized by our gate-sizing routine. Since the dynamic program is mainly about deciding the logic structure of the circuit, while all further decisions will be made later, it is sensible to assume at this point that gate-sizing and repeater tree construction will lead to "typical" results, i.e. the delays can be approximated by characteristic values. Our approach is therefore to have a realistic but simple-to-use delay estimate during the first part of the algorithm and combine it with powerful detailed optimization routines that yield accurate final slack values computed by the static timing engine (Subsection IV-E).

The bottom-up functioning of the dynamic program leads to similar considerations for our placement policy. During the algorithm, a partial solution function will be required at different locations because it contributes to different other functions at different levels of the dynamic program, not all of which will contribute to the complete function that is eventually built. Only at the last level will it become clear at which locations the partial function signals were really needed, but then the locations chosen earlier will already have influenced decisions taken later on until the last level. Therefore, whenever we place a new gate on the chip area, our placement decision will depend on the input signals that feed the new gate, but not on the next gates because we do not know which of them will be significant. On this basis, our best guess is to choose a location that results in an earliest possible arrival time for the new signal.

To account for the time a signal needs to reach the next gate, we model wire delay by multiplying distances by a delay-per-distance parameter d_{dist}. This is realistic because the delay of a buffered connection grows linearly with distance. We can think of a signal as travelling on the chip area and merging with another signal at a new gate (Fig. 2). The two signals start from their respective locations at their respective arrival times and run towards each other until they eventually meet. Since the gate delay is assumed to equal d_{gate} for both input pins, the arrival time at the output of the new gate is minimized if we place it between its predecessors such that the two input signals arrive simultaneously. Note that arrival times and locations of the original subpath input signals are known from static timing analysis and the design data. The d_{dist} parameter also accounts for inverter delay from buffering long wire connections. We obtain the value of d_{dist} by analyzing long-distance repeater chains as described in [1].

We are now ready to describe how the algorithm generates

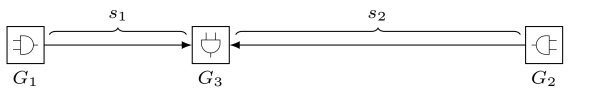


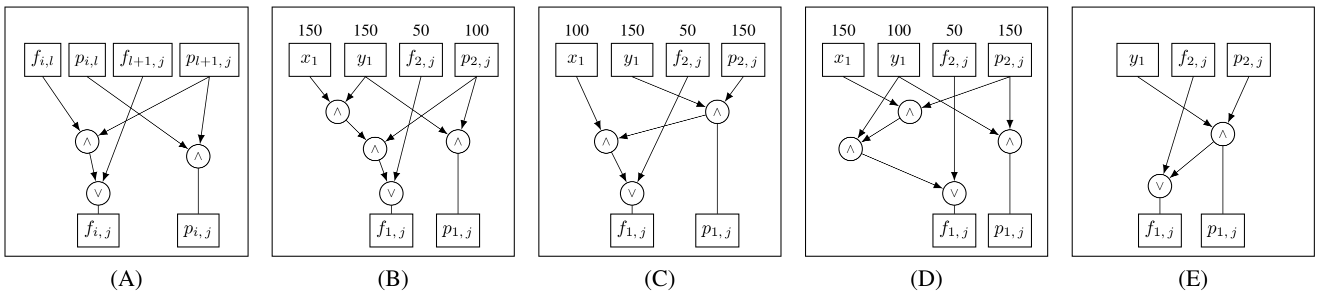
Fig. 2. Placement of a new gate G_3 fed by gates G_1 and G_2. Starting at time t_i, the output signal of G_i reaches G_3 at time $t_i + d_{\text{dist}} s_i$ if G_3 is at distance s_i from G_i ($i = 1, 2$). If $|t_1 - t_2| \leq d_{\text{dist}}(s_1 + s_2)$, then the position of G_3 is chosen such that the two signals arrive at the same time, leading to an arrival time of $t_3 = t_1 + d_{\text{dist}} s_1 + d_{\text{gate}} = t_2 + d_{\text{dist}} s_2 + d_{\text{gate}}$ for the output signal of G_3. Otherwise, G_3 is placed directly at G_1 (for $t_1 > t_2$) or G_2 (for $t_2 > t_1$); this case is common in practice.

the input signals to the function f defined in (1). When an instance is prepared for the dynamic program, main-path inputs are grouped in AND and OR sets. Gates computing the respective multiway function for each set are built by the following greedy algorithm: as long as there is more than one signal in the set, take the two earliest signals and combine them by an AND2 or OR2 gate; replace the two signals by the output of the new gate and repeat until the set consists of just one signal. The locations and arrival times of the new gates are determined as described above (using d_{dist} and d_{gate}), just like the gates created subsequently by the dynamic program. After building up gates for the multiway functions, $x_1, y_1, \ldots, x_n, y_n$ (the inputs for f) are available.

C. Running the Dynamic Program

The dynamic programming algorithm starts from input signal pairs (x_i, y_i), which are constructed as described in the previous subsections, and builds up a circuit that computes $f(x_1, y_1, \ldots, x_n, y_n)$. It maintains a table whose kth row stores pointers to those gates that compute the partial functions $(f,p)_{i,\, i+k-1} := (f_{i,\, i+k-1},\, p_{i,\, i+k-1})$ corresponding to an interval of k input variable pairs. For $k = 1$, these would be $(f,p)_{i,i} = (x_i \wedge y_i,\, y_i)$; however, to avoid circuits that can be readily reduced by trivial transformations, we set up the first row with the original pairs (x_i, y_i) and treat them in a special way (see below). Then, for $k = 1, \ldots, n-1$, after rows 1 up to k have been completed, row $k+1$ is filled with the functions $(f,p)_{i,\, i+k}$ for all $i \in \{1, \ldots, n-k\}$. The solution for $(f,p)_{i,\, i+k}$ is determined by finding an appropriate $l \in \{i, \ldots, i+k-1\}$ such that the arrival time (according to our two-parameter delay model) of the gate computing $f_{i,\, i+k} = (f_{i,l} \wedge p_{l+1,\, i+k}) \vee f_{l+1,\, i+k}$ is minimized. (This arrival time dominates the arrival time of the gate computing $p_{i,\, i+k}$, as can be proved by induction.) The new subcircuit is shown in Fig. 3 (A). The last row will have exactly one entry, namely for the complete function pair $(f,p)_{1,n}$.

Care is necessary for the special case of initial pairs (x_i, y_i). If we started with pairs $(f,p)_{i,i} = (x_i \wedge y_i,\, y_i)$ instead of (x_i, y_i), then $(f_{1,1} \wedge p_{2,2}) \vee f_{2,2} = ((x_1 \wedge y_1) \wedge y_2) \vee (x_2 \wedge y_2)$ would be built, for example, although $((x_1 \wedge y_1) \vee x_2) \wedge y_2$ is more efficient. An example for combining an initial (x,y) pair with an (f,p) pair on its right is given in Fig. 3 (B)–(D). We must also take dummy variables into account here, since x_1 and y_n might stand for a dummy OR or AND set, respectively (Fig. 3 (E)).



$f_{i,l}$ $p_{i,l}$ $f_{l+1,j}$ $p_{l+1,j}$
$f_{i,j}$ $p_{i,j}$
(A)
150 150 50 100
x_1 y_1 $f_{2,j}$ $p_{2,j}$
$f_{1,j}$ $p_{1,j}$
(B)
100 150 50 150
x_1 y_1 $f_{2,j}$ $p_{2,j}$
$f_{1,j}$ $p_{1,j}$
(C)
150 100 50 150
x_1 y_1 $f_{2,j}$ $p_{2,j}$
$f_{1,j}$ $p_{1,j}$
(D)
y_1 $f_{2,j}$ $p_{2,j}$
$f_{1,j}$ $p_{1,j}$
(E)

Fig. 3. Combining pairs in a dynamic programming step. The regular case is shown in (A): for $1 \le i < l < j-1 < n$, pairs $(f,p)_{i,l}$ and $(f,p)_{l+1,j}$ are combined to a new pair $(f,p)_{i,j} = \big((f_{i,l} \wedge p_{l+1,j}) \vee f_{l+1,j},\ p_{i,l} \wedge p_{l+1,j}\big)$. Initial pairs are treated as a special case; as an example, (B)–(D) show how the leftmost pair (x_1, y_1) is combined with a partial solution $(f,p)_{2,j}$ for some $j \in \{3,\ldots,n\}$. Since $(f,p)_{1,1}$ would be equal to $(x_1 \wedge y_1,\, y_1)$ if we had built it explicitely for the first row of the dynamic program, the straightforward solution for $(f,p)_{1,j}$ is $((f_{1,1} \wedge p_{2,j}) \vee f_{2,j},\ p_{1,1} \wedge p_{2,j}) = (((x_1 \wedge y_1) \wedge p_{2,j}) \vee f_{2,j},\ y_1 \wedge p_{2,j})$. This solution is indeed tried (B). While the product $p_{1,j}$ is always built as $y_1 \wedge p_{2,j}$, the function $f_{1,j}$ could also be realized as $(x_1 \wedge p_{1,j}) \vee f_{2,j}$ (C) or as $((x_1 \wedge p_{2,j}) \wedge y_1) \vee f_{2,j}$ (D). The numbers above the signals give the respective gate-delay sums for an example d_{gate} value of 50 picoseconds: e.g., both x_1 and y_1 traverse three new gates until $f_{1,j}$ is reached in (B), etc. None of the three boxes has a gate-delay profile that dominates the two others, so each solution may be the best choice for some arrival time pattern. Wire delays (modelled by parameter d_{dist}) are ignored in the diagrams, but they do add to the arrival times in the actual dynamic program. Among the three distinct solutions, the one with the best arrival time for $f_{1,j}$ is chosen. Of course, this will only be fixed as a solution for $(f,p)_{1,j}$ in row j if there is no other $l \in \{2,\ldots,j-1\}$ such that combining $(f,p)_{1,l}$ and $(f,p)_{l+1,j}$ (or (x_j, y_j) if $l = j-1$) gives a better solution. If x_1 is representing a dummy OR set, then (x_1, y_1) and $(f,p)_{2,j}$ are always combined as $(f,p)_{1,j} = (p_{1,j} \vee f_{2,j},\ y_1 \wedge p_{2,j})$ since this is the most efficient solution when x_1 is set to 1 (E). For other cases, similar case distinctions are made, e.g. when combining two initial pairs (x_i, y_i) and (x_{i+1}, y_{i+1}) (not shown here).

D. Mapping the Solution Logic into the Given Technology

While filling the dynamic programming table, we have constructed (and placed) a circuit that computes a lot of unused partial functions and contains gates whose pins may carry inversions. Next we trace backwards from the final gate for $f_{1,n}$ and identify its fanin-cone. Starting at this gate, we use De Morgan's rule to push inversions over the gates such that only NAND2 and NOR2 gates are used. This process includes the side logic gates and ends at the original inputs, where inverters are built as needed. As gate fanouts can be greater than one, inverters may also have to be built at the output pins of gates that drive input pins of opposite parity.

So far, all computations could be done appropriately in custom data structures without changing the actual netlist of the chip. After identifying the relevant logic and performing the De Morgan sweep, we modify the netlist itself by creating the gates at the locations determined by the dynamic program and connecting them with one another and the existing logic on the chip. Some of the original gates on the critical path might feed other gates besides the next gate on the path. We retain the starting section of the path up to the gate that drives the last of these side outputs. The tail of the path is deleted.

E. Detailed Optimization of the Solution Logic

The circuit we have constructed computes the correct Boolean function and has been tailored by the dynamic program to minimize the arrival time of the gate computing the complete function. Even so, electrical optimization is very important to find a netlist with improved slack according to static timing analysis. In our implementation, we call a gate-sizing routine [2] on all newly created gates that uses the detailed delay model of the static timing engine. Next we rebuild repeater trees on the new logic [1]; this cannot be done before gate-sizing because it would then be based on unrealistic slacks. After tree optimization, a second gate-sizing call brings the new logic to its final form. This discloses the actual slack improvement for the path instance (up to degradation incurred by placement legalization).

We point out that optimization techniques such as pin swapping, wire type assignment, V_t level assignment, localized logic changes etc. have not yet been included in our implementation. These would offer additional optimization opportunities beyond gate-sizing and repeater tree construction.

F. Iterative Application of the Algorithm

From looking at the performance guarantee given in Theorem 1, we might expect that the longer the path is that we restructure in the dynamic program, the more we gain in terms of slack. However, it turns out that optimizing just a part of a resolvable section can give better slack improvements in practice. This can have several reasons. Although the constants in the theorem are small, a signal that has to pass through additional gates may take more overall time than in the original path. During the dynamic program, we do not know in advance where inverters will be needed to ensure Boolean correctness, and due to their discrete nature, the delay-per-distance parameter that accounts for them may give inaccurate estimates. For some arrival time profiles, there might not be good split positions at all recursive stages (the two parts in (2) are more likely to minimize the arrival time of the combination function if the arrival times of the parts are balanced). Properties of the technology library have a strong impact on how slack-optimal netlists look like, and arrival time estimates in the dynamic program may be less accurate when signals have to traverse a larger number of new gates, which is more likely when long paths are optimized. So the distribution of the input arrival times, combined with discrete effects of the technology library and the electrical characteristics of the gate-sizing and repeater tree instances, can bias the optimization outcome towards partial instances. Indeed, even a path consisting of one complex gate (a multiplexer, for example) could be worth restructuring when the library provides just a small range of choices for the complex cell and a lot of logically equivalent simple cells such as NAND2 with better optimization opportunities for the gate-sizing routine.

In our implementation, we consider all subpaths of a resolvable section that do not consist entirely of repeaters. Each of these produces up to four problem instances for the dynamic program, depending on the choices of the inversions during path normalization (Subsection IV-A) and CMOS mapping (Subsection IV-D). This approach makes it necessary to keep a record of the tentative changes we make because the chip must be reverted to its original state before the next candidate solution is processed.

The quality of a solution circuit C is measured by the slack improvement δ_{path} at the gate computing the overall function because this slack is the worst slack of all paths through C. However, as the capacitance at some inputs to C can increase, the slack due to a side path outside C at a repeater tree root might get worse even if all the slacks within C improve. Since we are working on the critical path, we allow side path slacks to get worse, but we reject instances where they drop below the original slack at the output gate (before C was inserted). This ensures that the overall worst slack on the chip can only improve (cf. Fig. 5).

The running time of our code is largely dominated by electrical optimization: gate-sizing and repeater tree construction take 80 to 95 per cent of the time, while running the dynamic programs is fast. To reduce the number of gate-sizing calls, we first run the dynamic program for all subinstances and sort the results in descending order according to the difference between the original arrival time and the arrival time estimated by the dynamic program, which is an estimate for the slack improvement δ_{path}. Then we continue with electrical optimization for just a few instances from the beginning of the sorted list and pick the best solution we have found. This usually yields the best possible slack improvement over all subinstances. All other solutions are discarded.

After optimizing the critical path, an incremental invocation of static timing analysis reveals the new critical path. We repeat the whole procedure described so far to improve the worst slack of the chip, working on the new critical path in each iteration. If the first few critical paths have very similar slack values, optimizing the worst path will not improve the overall worst slack much even if a large δ_{path} could be achieved. However, iterating critical path optimization can improve the worst slack by more than the average δ_{path} value.

V. Experimental Results

Improvements achievable by restructuring long paths clearly depend on prior optimization: fortunate decisions during early logic synthesis will make slack gains harder, whereas poor physical design will offer easy optimization opportunities for

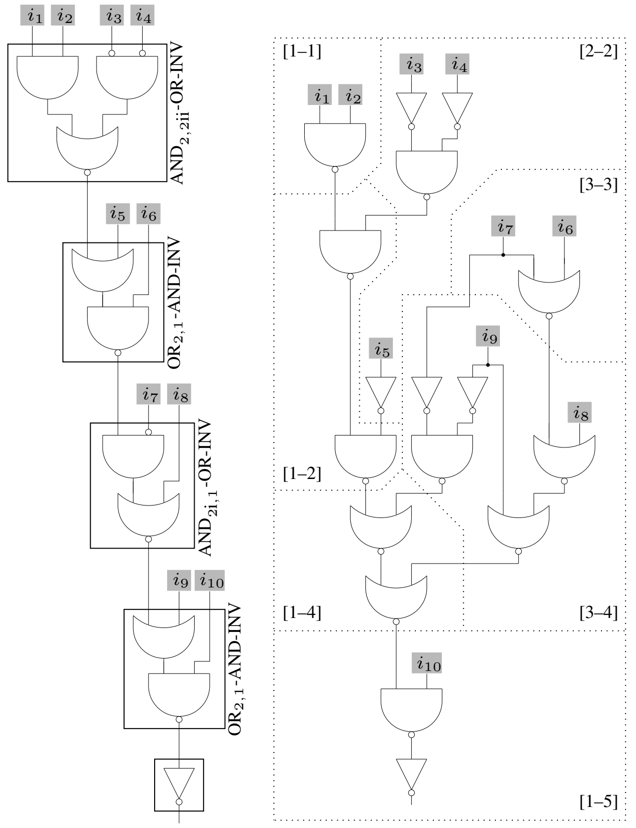


Fig. 4. Subfunction grouping in the dynamic program (Dirk, iteration 17). The path on the left resolves to five pairs of main-path inputs with dummy sets at both ends ($f = ((\ldots(x_1 \wedge y_1) \vee \ldots) \vee x_5) \wedge y_5$ with $x_1 = 1,\ y_1 = i_1 \wedge i_2,$ $x_2 = \neg i_3 \wedge \neg i_4,\ y_2 = \neg i_5,\ x_3 = \neg i_6,\ y_3 = \neg i_7,\ x_4 = i_8,\ y_4 = \neg i_9,$ $x_5 = \neg i_{10}$ and $y_5 = 1$). The circuit on the right is the solution of the dynamic program and has been applied to the netlist in this iteration. Dashed lines separate sections of the circuit that correspond to different levels of the dynamic program, as indicated by variable pair intervals. For example, the part labelled [3–4] computes $f_{3,4}$ and $p_{3,4}$ from (x_3, y_3) and (x_4, y_4).

any tool that incorporates gate-sizing and repeater tree construction as subroutines. Since detailed electrical optimization is necessary for building up a netlist with meaningful slacks (Subsection IV-E), we cannot evaluate logic optimization out of context. In order to give a fair assessment of our algorithm, we have made sure that our code uses the same gate-sizing and repeater tree routines with the same parameter settings that are used in earlier optimization steps.

For the sake of comparability, we have prepared each design in the same way. First, we have run a timing driven loop tool with standard parameter settings [2], which runs two rounds consisting of placement and timing-optimization, with timing-driven netweights in the second round. Next, gate-sizing and repeater tree routines [1] have been iterated until the overall worst slack could not be improved further, and the designs have been legalized. This is the starting point for our code. Naturally, worst slack levels after the preparatory optimization steps vary.

As indicated in Subsection IV-F, our algorithm is iterated and works on the current critical path in each iteration. We

TABLE I
TESTBED CHARACTERISTICS

Chip	Technology	Number of gates	Slack target[a]	Worst slack[b]	Slack sum[c]	Standard-cell density	Critical path clock cycle
Jens	180 nm	64 K	1.250 ns	1.014 ns	–2016 ns	0.7134	7.0 ns
Fazil	130 nm	64 K	–2.000 ns	–2.499 ns	–2230 ns	0.7493	5.0 ns
Franz	90 nm	69 K	0.300 ns	0.186 ns	–828 ns	0.7488	4.8 ns
Lucius	65 nm	72 K	0.000 ns	–0.299 ns	–2553 ns	0.3373	2.7 ns
Dirk	65 nm	98 K	0.600 ns	0.284 ns	–7872 ns	0.3688	3.0 ns
Heidrun	130 nm	297 K	0.250 ns	–0.389 ns	–5376 ns	0.6447	15.0 ns
Wilhelm	130 nm	1361 K	–1.200 ns	–1.690 ns	–7906 ns	0.4048	3.2 ns, 4.0 ns
Karsten	130 nm	3118 K	0.300 ns	–0.151 ns	–7130 ns	0.4817	1.7 ns
Herbert	90 nm	3447 K	–2.000 ns	–2.739 ns	–242 ns	0.8529	1.6 ns, 4.7 ns
TRIPS	130 nm	5920 K	–1.500 ns	–2.071 ns	–3393 ns	0.5756	4.5 ns

[a] Used to compute slack sum and to control gate-sizing and repeater tree construction.
[b] Legal placement.
[c] Sum of slacks below target (shifted by target).

TABLE II
OPTIMIZATION RESULTS

Chip	Logic optimization					First legalization		Second legalization		**Worst-slack gain**[b]	Density incre-ment[c]
	number of iter.	aver. time per iter.[a]	average δ_{path}	worst slack	slack sum	worst slack	slack sum	worst slack	slack sum		
Jens	200	35 s	91 ps	1.114 ns	–1340 ns	0.954 ns	–1918 ns	1.053 ns	–1486 ns	39 ps	0.0085
Fazil	200	40 s	155 ps	–2.320 ns	–1585 ns	–2.498 ns	–2575 ns	–2.356 ns	–1914 ns	143 ps	0.0081
Franz	200	133 s	108 ps	0.245 ns	–345 ns	0.204 ns	–620 ns	0.246 ns	–186 ns	60 ps	0.0078
Lucius	200	30 s	109 ps	–0.084 ns	–591 ns	–0.245 ns	–754 ns	–0.069 ns	–354 ns	230 ps	0.0062
Dirk	200	19 s	179 ps	0.377 ns	–7202 ns	0.336 ns	–7418 ns	0.384 ns	–6480 ns	100 ps	0.0042
Heidrun	200	162 s	108 ps	–0.259 ns	–5217 ns	–0.265 ns	–5208 ns	–0.260 ns	–4653 ns	129 ps	0.0018
Wilhelm	200	220 s	119 ps	–1.338 ns	–2682 ns	–1.610 ns	–4052 ns	–1.364 ns	–612 ns	326 ps	–0.0005
Karsten	31	87 s	26 ps	–0.032 ns	–7140 ns	–0.172 ns	–7082 ns	–0.052 ns	–6593 ns	99 ps	0.0000
Herbert	64	154 s	168 ps	–2.251 ns	–162 ns	–2.306 ns	–225 ns	–2.308 ns	–152 ns	431 ps	–0.0003
TRIPS	50	490 s	173 ps	–1.583 ns	–212 ns	–1.625 ns	–416 ns	–1.539 ns	–58 ns	532 ps	0.0000

[a] Running times on a 2.6GHz AMD Opteron. For Jens and Fazil, a 600MHz IBM PowerPC RS64III has been used and running times have been scaled down (4x).
[b] Improvement of worst slack after final legalization over initial worst slack as in Table I.
[c] Adding this number to the standard-cell density of Table I gives the standard-cell density after final legalization.

use IBM's EinsTimer engine for static timing analysis with interconnect delay modelled by Steiner trees and Elmore delay. The number of iterations has been limited to 200, but can be smaller if no further improvement is achieved. In each iteration, all dynamic programming solutions were ordered before the first gate-sizing call by expected slack improvement, i.e. the estimates for δ_{path} of the dynamic programs. Since running times are dominated by gate-sizing (including incremental static timing analysis), we use the following scheme to cut down the number of detailed optimization instances. Starting with an initial target of 150 ps for δ_{path}, we optimize the solution candidates one by one and select the best solution as soon as the target is reached. The δ_{path} target is decreased by 10 ps each time five candidates have been processed. This way, we may find the best solution even if it is not the first candidate and even if we do not know which δ_{path} value can be achieved. We remark that it is possible to reduce running times further by optimizing gate-sizing instances in parallel.

When copying a dynamic programming solution to the netlist, we use medium-sized cells for NAND2, NOR2 and INV as starting points for gate-sizing. If there is a choice among different V_t levels, we use the level that is predominant on the critical path. Technology library analysis yields characteristic values for the delays of the NAND2 and NOR2 cells in an optimized netlist with typical input slews and capacitative loads; the average of the two is used as an initial value for the d_{gate} parameter mentioned in Subsection IV-B. The d_{dist} parameter is initialized with the average per-nanometer delay of an optimized inverter chain, again obtained from library analysis [1]. After each iteration, the average delay through the NAND2 and NOR2 gates on the critical path of the newly inserted logic is determined (gate-sizing often slows down gates off the critical path), and d_{gate} is set to 90% of its current value plus 10% of that average delay.

In our experiments, logic optimization is followed by placement legalization. To compensate for slack degradation, a limited amount of gate-sizing is performed. Then a second legalization step brings the netlist to its final state.

Our testbed consists of recent industrial IBM ASIC designs; we are grateful to our cooperation partner IBM for providing the data. We also thank the Department of Computer Sciences at the University of Texas at Austin for granting us access to the TRIPS data [14]. The other chip names have been changed for confidentiality.

Characteristics of the testbed chips can be found in Table I. To control the gate-sizing and repeater tree routines after the

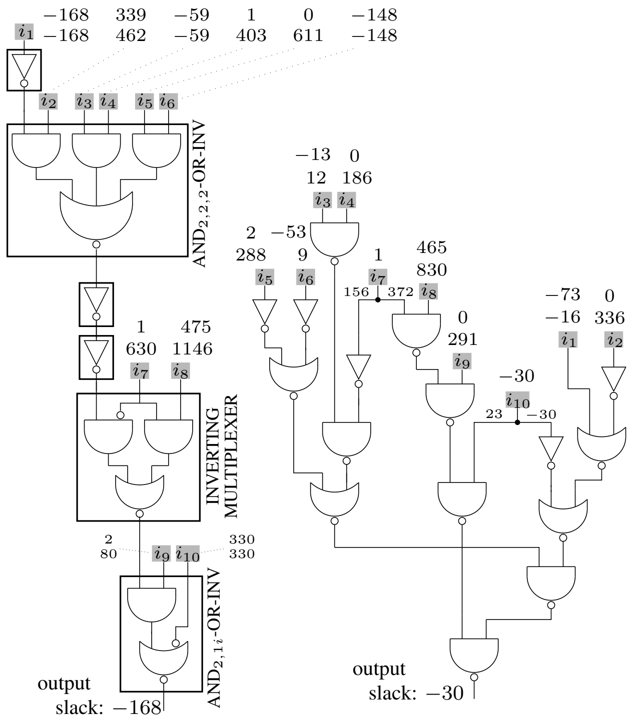


Fig. 5. Slacks (in picoseconds) before and after path restructuring (Lucius, iteration 38). The path on the left has been replaced by the circuit on the right, giving a slack improvement of $\delta_{\text{path}} = (-30\,\text{ps}) - (-168\,\text{ps}) = 138\,\text{ps}$. Two numbers are specified at each input signal. The bottom number shows the slack at the gate input pin fed by the signal. Note that the smallest of these numbers is always equal to the output slack of the circuit. The top number is the slack at the root of the repeater tree that contains the input pin, which can be smaller due to side paths through the root. The original slack at the interlinked input pin (fed by i_1) climbs from -168 ps to -16 ps in the new logic. The new critical input is i_{10}; there are two paths from this input to the overall function output with respective slacks of -30 ps (critical path) and 23 ps. The smallest root slack of the original path always equals the overall worst slack of the chip. In this example, the smallest root slack after the replacement (-73 ps) is again found at i_1. The algorithm would have rejected the restructuring solution if any root slack had dropped below -168 ps. Note how the logic has been rebalanced so that i_1 and i_6 (the two worst inputs originally) traverse only four NAND/NOR stages.

timing-driven loop and during our algorithm, an individual slack target has been chosen for each chip. However, logic optimization itself ignores the slack target. It is also used for computing a number called slack sum, which is the sum of output pin slack minus the target, summed over all output pins below the target. Next to worst slack, the slack sum is a quality measure for our optimization results. The standard-cell density is the ratio of the area taken up by standard cells and the placement area not taken up by macros and blockages. Table I also gives the cycle times of the clocks seen on the critical paths that have been optimized by our algorithm.

Examples for paths that have been restructured are given in Fig. 4 and 5; a summary of the results can be found in Table II. The second column from the right shows that our code successfully improves the overall worst slack and achieves gains of several hundred picoseconds on some chips. We emphasize that these improvements are made independently of optimization techniques such as those mentioned in the introduction, which can give additional slack gain.

As can be expected for a testbed of different industrial designs, the degree varies to which logic optimization can improve timing. Comparing the slack sums after final legalization to their values before logic optimization, we see that the overall quantity of timing violations is considerably reduced on all chips. Even when worst slack gains are small, reducing the slack sum is important.

Table II also gives worst slack and slack sum numbers at intermediate optimization stages. Naturally, slack figures deteriorate by legalization just after logic optimization. However, some gate-sizing (not even with highest tool effort) makes good the loss, even when followed by another legalization.

For a tool that creates new circuits and changes gate sizes, it is important to keep chip density under control. The last column of Table II shows that the impact of our tool on density is harmless. In fact, density savings are possible on some chips.

Summarizing the results, we have demonstrated that our implementation is a powerful logic optimization tool that can improve worst slack on real-world designs by up to several hundred picoseconds.

References

[1] C. Bartoschek, S. Held, D. Rautenbach, J. Vygen, *Efficient Generation of Short and Fast Repeater Tree Topologies*, International Symposium on Physical Design (2006), 120–127.

[2] S. Held, *Fast Gate Sizing and Timing Closure for Multi-Million Cell Designs*, Report No. 07969, Research Institute for Discrete Mathematics, University of Bonn, 2007.

[3] B. Korte, D. Rautenbach, J. Vygen: *BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip*, Proceedings of the IEEE **95** (2007), 555–572.

[4] J. Liu, S. Zhou, H. Zhu, C.-K. Cheng, *An Algorithmic Approach for Generic Parallel Adders*, International Conference on Computer-Aided Design 2003, 734–740.

[5] V. G. Oklobdzija, D. Villeger, S. S. Liu, *A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach*, IEEE Transactions on Computers **45** (1996), 294–306.

[6] D. Rautenbach, C. Szegedy, J. Werber, *Delay optimization of linear depth boolean circuits with prescribed input arrival times*, Journal of Discrete Algorithms **4** (2006), 526–537.

[7] D. Rautenbach, C. Szegedy, J. Werber, *Asymptotically Optimal Boolean Circuits for Functions of the Form* $g_{n-1}(g_{n-2}(\ldots g_3(g_2(g_1(x_1,x_2),x_3),x_4)\ldots,x_{n-1}),x_n)$ *given Input Arrival Times*, Report No. 03931, Research Institute for Discrete Mathematics, University of Bonn, 2003.

[8] J. E. Savage, *Models of Computation: Exploring the Power of Computing*, Addison-Wesley Longman, Reading, MA, 1998.

[9] P. F. Stelling, V. Oklobdzija, *Design strategies for the final adder in a parallel multiplier*, 29th Asilomar Conference on Signals, Systems and Computers (1995), 591.

[10] P. F. Stelling, V. Oklobdzija, *Design strategies for optimal hybrid final adders in a parallel multiplier*, Journal of VLSI Signal Processing Systems **14** (1996), 321–331.

[11] I. Sutherland, B. Sproull, D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, Morgan Kaufmann, San Francisco, CA, 1999.

[12] L. Stok et al., *BooleDozer: Logic synthesis for ASICs*, IBM Journal of Research and Development **40** (1996), 407–430.

[13] L. Trevillyan, D. Kung, R. Puri, L. N. Reddy, M. A. Kazda, *An Integrated Environment for Technology Closure of Deep-Submicron IC Designs*, IEEE Design & Test of Computers **21** (2004), 14–22.

[14] http://www.cs.utexas.edu/users/cart/trips/index.html.

[15] W.-C. Yeh, C.-W. Jen, *Generalized Earliest-First Fast Addition Algorithm*, IEEE Transactions on Computers **52** (2003), 1233–1242.