Layout Driven Retiming Using the Coupled Edge Timing Model

Ingmar Neumann and Wolfgang Kunz, *Member, IEEE*

***Abstract*—Retiming is a widely investigated technique for performance optimization. It performs powerful modifications on a circuit netlist. However, often it is not clear whether the predicted performance improvement is still valid after placement has been performed. This paper presents a new retiming algorithm using a highly accurate timing model. It takes into account the effect of retiming on capacitive loads of single wires as well as fanout systems. Further, we propose the integration of retiming into a timing-driven standard cell placement environment. Retiming is used as an optimization technique throughout the whole placement process. The experimental results show the benefit of the proposed approach. In comparison with the conventional design flow based on the standard FEAS algorithm, our approach achieved an improvement in cycle time of up to 34% and 17% on the average.**

***Index Terms*—Interconnect delay, logic-layout interaction, placement, retiming.**

I. Introduction

RETIMING, originally proposed by Leiserson and Saxe [1], [2], is a powerful and well-known technique for performance optimization of digital circuits. It is based on relocating registers while preserving the functionality of the circuit. Many improvements and extensions to the original ideas have been developed, like acceleration techniques [3] dramatically speeding up execution time, concepts for integrating retiming into logic synthesis [4], algorithms for retiming level clocked circuits [5], [6], algorithms taking register setup and hold times into account [7], [8], algorithms for retiming registers with enable inputs [9], and as algorithms that can improve testability [10].

When optimizing large sequential circuits the use of retiming is very attractive. Other sequential optimization techniques like state re-encoding suffer from the state explosion problem and usually fail for circuits containing more than a few hundred registers. Retiming on the other hand does not require an explicit representation of the state set. It operates directly on a netlist description of the circuit and can handle circuits with thousands of registers.

Nevertheless, retiming has encountered only limited acceptance in industrial practice. This is mainly for two reasons. First, a retimed circuit is very hard to verify against the original circuit.

However, in recent years there have been advances in the area of sequential equivalence checking for retimed circuits [11], [12]. There is the promise that industrial tools capable of verifying retimed circuits are becoming available. This paper deals with a second problem affecting the acceptance of retiming in practice. The choice of an accurate timing model in combination with an appropriate retiming algorithm is a delicate issue.

Manuscript received July 8, 2002; revised October 28, 2002. This paper was recommended by Associate Editor S. Hassoun.

The authors are with the Department of Electrical Engineering and Information Technology, University of Kaiserslautern, D-67653 Kaiserslautern, Germany.

Digital Object Identifier 10.1109/TCAD.2003.814253

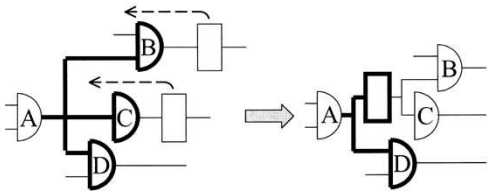

Fig. 1. Retiming sink gates of a multisink net.

With conventional timing models and retiming techniques, it often remains unclear whether the predicted performance improvement is still valid after placement has been performed.

The original *FEAS* algorithm, developed by Leiserson and Saxe, finds retiming for a circuit such that a given cycle time is met provided such a retiming exists. It is based on a simple timing model assuming gate delays to be load independent. Unfortunately for CMOS technology, this model is not accurate enough as gate delays cannot be considered to be load independent, and retiming registers change the loads of the gates.

In [13]–[15], more sophisticated timing models are used. For each edge in the retiming graph, several delay values are calculated covering the two cases that this edge can contain zero or at least one register. This is already a strong improvement over previous models. However, these models do not correctly describe situations given at fanout trees, as shown in Fig. 1.

In real circuits, retiming gates B and C change the load (drawn in bold lines) seen by gate A and, therefore, also changes the delay of gate A. This, however, not only affects the data arrival time at gates B and C, but also at gate D.

In practice, retiming of registers into fanout trees may change the topology of the affected nets dramatically and can change arrival times even on paths where no registers have been moved. Ignoring this effect may lead to unpredictable results.

The advent of deep-submicron technologies introduced additional difficulties by increasing the influence of wire length on the total delay. Loads resulting from wires are affected by retiming even more than loads resulting from gate inputs and, above all, are not known before placement.

An interesting approach for integrating retiming into the design flow was presented in [16]. Retiming is coupled with partitioning-based floor planning, allowing performance optimization during an early physical design stage. However, the limited accuracy of the wire-length data being available during floor planning limits the accuracy of the delay calculations.

An approach to integrate retiming into detailed placement was presented in [17]. After performing a conventional placement and routing procedure, an optimization loop consisting of wire-length estimation, retiming, and register insertion is entered. Even though this approach produces promising results, it does not fully exploit the potential of coupling placement and retiming. Note that a timing-driven placer aggressively tries to shorten wires on critical paths while paying less attention to less critical wires. This can lead to a balance of path lengths reducing the optimization potential for retiming.

In order to take all of the above into account, we propose a new approach for making retiming more practical. It consists of the following two parts. The first part consists of a new timing model being used in the retiming graph and a new retiming algorithm. The timing model allows an accurate modeling of load changes in fanout systems caused by retiming. The retiming algorithm is able to exploit the increased optimization potential resulting from the improved accuracy while preserving the polynomial time complexity of the original *FEAS* algorithm. Furthermore, we propose a tight coupling of retiming and simulated annealing-based detailed placement. This approach does not use retiming for a postplacement optimization, but employs it as an optimization technique throughout the whole placement process.

II. Retiming With Accurate Timing Model

A powerful and efficient retiming algorithm for cycle time minimization is *FEAS* [1], [2]. The *FEAS* algorithm has many attractive properties and, therefore, we wish to adopt the general *FEAS* strategy also in this work. However, combining the basic idea of *FEAS* with an accurate timing model is a delicate issue. The difficulty arises from the fact that *FEAS* is based on an assumption called *path delay monotonicity constraint* in [14]. It is assumed that for any path of the circuit the data arrival time at a register can never grow if the register is moved backward in the circuit. The simple timing model of the original *FEAS* always fulfills this assumption. Unfortunately, this can change if more complex timing models are used. If we consider the wire loads in gate netlists mapped for typical standard cell libraries, it turns out that, in practice, the monotonicity assumption holds for two terminal nets in the vast majority of the cases. However, the problem occurs at the point where we model situations as shown in Fig. 1 more realistically. In such cases, the monotonicity assumption indeed may be violated. There is also a second problem with multisink nets. Even if monotonicity is not violated, the *FEAS* strategy of retiming a critical vertex often leads to suboptimal results. Note that shifting a register into one branch of a multiterminal net also affects the data arrival time on paths leading through the other branches of that net.

Trying to maintain the efficiency of *FEAS* on one hand and using a realistic timing model on the other hand requires a more sophisticated solution for retiming critical vertices.

Our new retiming algorithm follows the same strategy as the well known *FEAS*-algorithm. We perform an alternating sequence of running timing analysis and eliminating constraint violations locally by retiming critical vertices. However, since we use a more complex and accurate timing model, extensive modifications were necessary both of the arrival time calculation and of the strategy of deciding when to retime a vertex.

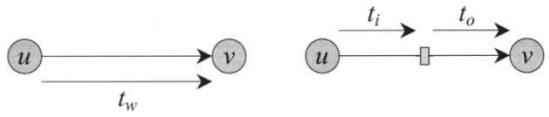

Fig. 2. Timing model for single sink net edge.

A. Circuit Model

The circuit is mapped onto a weighted directed graph $G = (V, E)$. Each logic gate is mapped onto a vertex v, being assigned a delay $t_d(v)$ and a retiming value $r(v)$ which is initially 0 and can be incremented during retiming. The logic gate corresponding to a vertex v is denoted by $g(v)$ in the following.

As in [2], G is extended by a host vertex representing the environment of the circuit.

A net driven by gate $g(u)$ with n driven gates $n \geq 1$ is modeled as a bundle of n edges ("branches") $B = \{b_i\} = \{(u, v_i) \subseteq E,\ 1 \leq i \leq n\}$. Each edge $e = (u, v) \in E$ is assigned a weight $w(e)$ denoting its initial number of registers. The number of registers on e during or after retiming is denoted by $w_r(e) = w(e) + r(v) - r(u)$. Furthermore, each edge is assigned several delay values as explained in the following sections.

1) Single Sink Net Edges: Nets with a single sink are modeled by edge bundles containing only one edge each. For these edges, our timing model is similar to the model proposed in [13]. An edge $e = (u, v)$ is assigned three delay values modeling two different cases as shown in Fig. 2.

- t_w: delay for a signal propagating from u to v in the case that there is no register on e, i.e., $w_r(e) = 0$;
- t_i: delay for a signal propagating from u to the data input of a register on e if there is at least one register on e, i.e., $w_r(e) > 0$;
- t_o: delay for a signal propagating from the output of a register on e to v, if $w_r(e) > 0$.

During data arrival time calculation, either t_i and t_o or t_w are used, depending on w_r. Data arrival time calculation is explained in more detail in Section II-C1.

2) Multisink Net Edges: A net with n sinks is modeled by a bundle of n edges $B = \{b_i\} = \{(u, v_i) \subseteq E,\ 1 \leq i \leq n\}$.

As explained in the introduction of the paper the delay of a particular branch of a multisink net (i.e., $n \geq 2$) depends on the register arrangements in the other branches of that net. To model these dependencies in the retiming graph, the delay values t_i, t_o, and t_w of an edge $b_i \in B$ have to depend on the weights of the other edges $b_j \in B$, $j \neq i$ belonging to the same bundle as b_i.

We handle this in our model by assigning three delay tables to each edge. Each table contains different values for t_i, t_o, or t_w, respectively, according to distinguishable register arrangements in B. Each table entry contains one delay value being valid for one particular register arrangement.

For distinguishing register arrangements, we have developed two models with different table sizes. This is elaborated in the following sections.

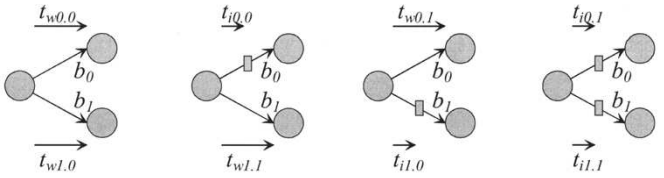

Fig. 3. Distinct register arrangements for a net with two sinks.

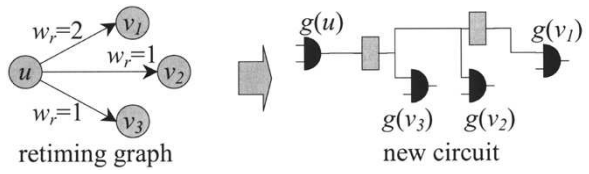

Fig. 4. Creating circuit from retiming graph.

a) Complex model: At first, we describe the more accurate complex model. In real circuits, driving different combinations of cells leads to different sums of gate-input capacitances. Furthermore, this requires nets of different topologies with different lengths. Our complex model enables us to take these effects into account so that circuit timing can be calculated in an accurate way.

Let us first consider the situation for the net at the inputs of the registers, i.e., the net being driven by $g(u)$. In the following, we denote the branch bundle representing this net by B. The t_w-values and t_i-values of the graph edges $b \in B$ depend on the load dependent portion of the delay of $g(u)$. The load being driven by $g(u)$ depends on the set of gates that $g(u)$ has to drive. The weights $w_r(b)$ of the graph edges determine which gates $g(v_i)$ belong to that set. These weights also determine whether $g(u)$ has to drive registers. Consequently, if we wish to consider all cases for the calculation of the load that $g(u)$ has to drive, we have to distinguish for each branch $b \in B$ whether $w_r(b) = 0$ or $w_r(b) \neq 0$. For B consisting of n branches we can enumerate 2^n distinct cases. For each of these cases we can calculate for every branch a t_i-value and a t_w-value, respectively. Since t_w is only defined for branches without registers and t_i only for branches with at least one register, this leads to 2^{n-1} entries for the t_w-and t_i-tables of each branch. The t_w-and t_i-tables consist of key-value pairs, where the key denotes an identifier for a register arrangement and the value denotes the corresponding t_w-and t_i-value, respectively.

Let us look at an example. Fig. 3 shows the graph representations for all possible register arrangements for a net with two sinks. Because the edge bundle consists of two branches, we can distinguish four cases. This leads to two different values for t_i and two different values t_w for each branch. Each register arrangement is given a binary number shown below the corresponding diagram. For illustration, in this example, the timing values are given two indices. The first index denotes the branch number and the second index denotes its position in the table. The vectors representing t_i delays are drawn a bit shorter than the t_w vectors to illustrate the fact that a register located on an edge is an endpoint of a signal path.

For the register arrangements shown in Fig. 3, the tables of timing values are as follows:

at branch b_0:

arr.	t_w
00	$t_{w0.0}$
10	$t_{w0.1}$

arr.	t_i
01	$t_{i0.0}$
11	$t_{i0.1}$

at branch b_1:

arr.	t_w
00	$t_{w1.0}$
01	$t_{w1.1}$

arr.	t_i
00	$t_{i1.0}$
11	$t_{i1.1}$

For each branch, we have one t_i-table and one t_w-table. Each table row represents a delay value being valid for a particular register arrangement. The second column contains the delay values itself, and the first column contains the number of the register arrangement for which the delay is valid.

Next, we consider the nets at the outputs of the registers and analyze the different cases for the values of t_o. We make the assumption that a bundle B is realized with a minimal number of registers (*register sharing*) when the netlist is created from the retiming graph. An example is shown in Fig. 4.

We believe this assumption to be reasonable because in typical standard cell libraries, latch cells have a much higher area requirement than simple logic cells. Therefore, if the driving force of a register is not sufficient to drive a long and widely spread net it is usually more effective to buffer the net and to replicate buffers rather than to replicate area consuming registers.

Consequently, for calculating the values t_o for a particular branch $b_i \in B$ we need to distinguish for each other branch $b_j \in B, i \neq j$, whether $w_r(b_i) = w_r(b_j)$ or $w_r(b_i) \neq w_r(b_j)$. This models the fact that under the assumption described above, two gates $g(v_i)$ and $g(v_j)$ with $w_r(b_i) = w_r(b_j)$ are driven by the same register. In the circuit shown in Fig. 4, this is the case for $g(v_2)$ and $g(v_3)$.

Because t_o values are only defined for edges with at least one register, we conclude that also for the tables for t_o we have to distinguish 2^{n-1} different cases leading to 2^{n-1} table entries.

b) Simple model: The complex model permits an accurate modeling of the real situation but it has the drawback of an exponential table growth. This makes it impossible to use this model for large fanout systems. Therefore, we propose a second model with linear growth of the table size.

To reach linear growth, we assume that the values of t_i and the values of t_w of a particular branch $b_i \in B$ are the same for all configurations containing the same number of branches b_j with $w_r(b_j) > 0, b_j \in B, i \neq j$.

In other words, for the calculation of the delay values of a particular branch, we only care about how many of the other branches of the bundle carry at least one register and not in which other branches these registers are located. This ignores the differences in the input capacitances of different gate types as well as the differences in the wire loads for the different register arrangements. Note, however, that the total load represented by a net and its sinks increases with the number of branches in this net. Therefore, the error resulting from ignoring the above-mentioned differences decreases for nets with a large number of branches. For nets with a small number of branches, we can afford to use the complex model.

Similarly, for the t_o-values of b_i, we assume that they are the same for all configurations containing the same number of branches b_j with $w_r(b_j) = w_r(b_i), b_j \in B, i \neq j$.

This limits the size of each table for each branch in a bundle B to $|B|$. In our implementation we use the complex model for

bundles with at most four branches. For the circuits examined in our experiments, it turned out that more than 90% of all nets can be described using the complex model. For the few larger nets with more than four sinks, we use the simple model.

B. Parameter Calculation

This section explains how the values of t_i, t_o, t_w, and t_d are determined from a circuit netlist.

The delay $t_d(v)$ of a vertex v denotes the load independent portion of the delay of gate $g(v)$. The calculation of the edge delays is based on a simple RC-model based approach

$$\text{delay} = \alpha \cdot C + \beta. \tag{1}$$

This model is widely used for delay calculation in cases where no detailed routing information is available, e.g., during placement. In the equation shown above, C denotes the capacitive load being driven by a particular gate, and α and β denote electrical parameters of the gate as they are derived from a cell library.

To calculate the edge delays t_i, t_o, and t_w of the outgoing edges of a vertex v, we have to determine the loads seen by $g(v)$ for all possible register arrangements. The load seen by $g(v)$ for a particular register arrangement consists of the input capacitances of the gates driven by $g(v)$ and the capacitance of the net connecting them.

In cases where no wire-length information is available, e.g., when retiming is performed during logic synthesis, wire capacitances may be assumed to have zero value.

In the following, we describe a wire-length-prediction method being applicable when placement data are available. This method is used for integrating retiming into placement as will be described in a later section.

The net lengths are predicted for each possible register arrangement using different methods for the simple and the complex delay model.

1) Complex Model: In this model, we assume specific register positions for the estimation of the net lengths. For the calculation of the t_i-and t_w-values in a branch bundle $B = \{b_i\} = \{(u, v_i)\}$, $1 \le i \le n$, the length $l(w)$ of net w driven by $g(u)$ is required. To estimate $l(w)$, we determine a (minimum size) rectangle R_0 containing $g(u)$ and those gates $g(v)$ that are driven by $g(u)$. If no register is present we use the half of the perimeter of R_0 as $l(w)$. If registers are present, we determine a rectangle R_1 containing the gates $g(v)$ that are driven by a register. If R_0 and R_1 overlap, we assume the register to be positioned inside R_0 and we use the half of the perimeter of R_0 as $l(w)$. If they do not overlap, we determine a minimum size rectangle R_r which touches R_0 and R_1. The center of R_r gives the assumed location for the register. As $l(w)$, we use the sum of the half perimeter of R_0 and the quarter perimeter of R_r. Fig. 5 shows the estimation of $l(w)$ for the register arrangements shown in Fig. 3. The binary number shown below the diagrams correspond to the numbers introduced in Fig. 3.

For a particular t_w and t_i, respectively, we obtain

$$t_w = \alpha \cdot \left(l(w) \cdot \text{cpl} + \sum_{\text{input ports connected to } w} c_{\text{in}} \right) \tag{2}$$

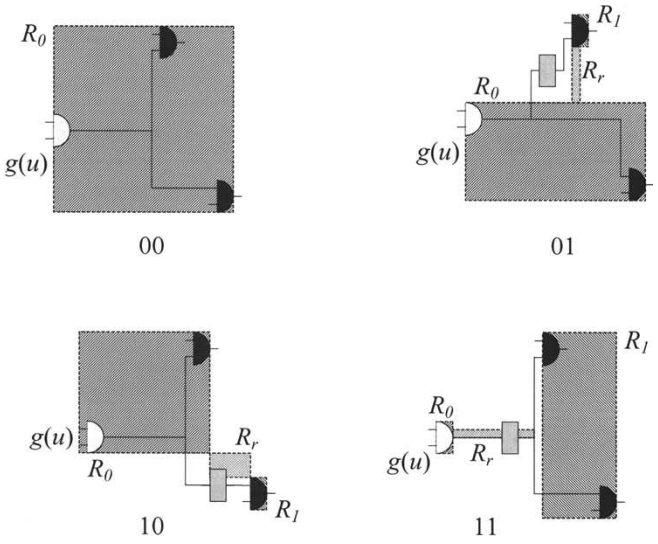

Fig. 5. Estimation of length $l(w)$ of a net w driven by $g(u)$ for different cases.

and

$$t_i = \alpha \cdot \left(l(w) \cdot \text{cpl} + \sum_{\text{input ports connected to } w} c_{\text{in}} \right) + t_{\text{setup}} \tag{3}$$

where α denotes the driving force of $g(u)$, *cpl* stands for the *capacitance per unit length*, c_{in} denotes the input capacitance of a port, and t_{setup} denotes the setup time of the register.

For the calculation of a t_o value of a branch for a particular register arrangement, the length $l(w')$ of net w' driven by a register is required. For the estimation of $l(w')$, we determine a rectangle R_0 containing all gates $g(v)$ that are driven by this register and a rectangle R_1 containing $g(u)$ and those gates $g(v)$ driven by $g(u)$ or by other registers. If R_0 and R_1 overlap, we assume the register to be positioned inside R_0 and we use the half of the perimeter of R_0 as $l(w')$. Otherwise, we determine a minimum size rectangle R_r touching R_0 and R_1 and assume the register to be located at the center of R_r. $l(w')$ is calculated using the sum of the half of the perimeter of R_0 and the quarter of the perimeter of R_r. For a particular t_o, we obtain

$$t_o = \alpha_{\text{reg}} \cdot \left(l(w') \cdot \text{cpl} + \sum_{\text{input ports connected to } w} c_{\text{in}} \right) + \beta_{\text{reg}} \tag{4}$$

where α_{reg} denotes the driving force and β_{reg} the load independent clock to output delay of the register.

2) Simple Model: As already mentioned in the simple model, we care only about how many branches of a bundle $B = \{b_i\} = \{(u, v_i)\}$, $1 \le i \le n$ carry a certain number of registers but not in which branches these registers are located. Therefore, we do not determine register positions for the length estimation. Instead we use a simplified length-estimation method using the half perimeter of the rectangle R containing $g(u)$ and all $g(v_i)$ with $(u, v_i) \in B$.

For the calculation of the t_i-and t_w-values of B, the length $l(w)$ of net w driven by $g(u)$ is required. Our estimation relies on the observation that the net length grows with increasing number

of gates $g(v)$ driven by $g(u)$. The more gates $g(v)$ are driven by registers the fewer gates need to be driven by $g(u)$. For a register arrangement containing m branches b_j with $w_r(b_j) = 0$, $0 \leq m \leq n$, we estimate $l(w)$ as follows:

$$l(w) = \frac{\text{perimeter}(R)}{2} \cdot \sqrt{\frac{m+1}{n+1}}. \tag{5}$$

In most cases, we observed that taking the square root gives more realistic values than a linear model.

For the load capacitance seen by $g(u)$, we obtain

$$c_{\text{load}} = l(w)^{*}\text{cpl} + \begin{cases} m \cdot c_{av}, & m = n \\ m \cdot c_{av} + c_{\text{reg}}, & m < n \end{cases} \tag{6}$$

where c_{av} denotes the average input capacitance of all input ports driven by $g(u)$ when no registers occur on any branch, i.e., $w_r(b_i) = 0$ for all $b_i \in B$. For $m < n$, at least one gate $g(v)$ is driven by a register. So, $g(u)$ has to drive one register in addition to the m logic gates and we have to add the input capacitance c_{reg} of the register. (Remember that we assume register sharing as described in Section II-A2a.

For a particular t_w and t_i, we obtain

$$t_w = \alpha \cdot c_{\text{load}} \text{ and } t_i = \alpha \cdot c_{\text{load}} + t_{\text{setup}}. \tag{7}$$

For determining a particular value of t_o, the length $l(w')$ of net w' driven by the register is required. For a particular branch $b_i \in B$, let m' denote the number of branches $b_j \in B$, $j \neq i$, showing the same register count as b_i, i.e., $w_r(b_i) = w_r(b_j)$. We calculate $l(w')$ as follows:

$$l(w') = \frac{\text{perimeter}(R)}{2} \cdot \sqrt{\frac{m'+1}{n+1}}. \tag{8}$$

The load seen by the register is

$$c_{\text{load}} = l(w') \cdot \text{cpl} + m' \cdot c_{av}. \tag{9}$$

The corresponding value of t_o results to

$$t_o = \alpha_{\text{reg}} \cdot c_{\text{load}} + \beta_{\text{reg}}. \tag{10}$$

C. Retiming Algorithm

Like in the *FEAS*-algorithm, we basically perform an alternating sequence of calculating arrival times and retiming critical vertices.

We have already explained that retiming one particular end vertex of a branch bundle has an influence on the delay of paths leading through other end vertices of the same bundle. We also showed that our coupled edge timing model enables us to take this effect into account. Now, we explain how to exploit the resulting optimization potential. For this purpose, we have to extend the basic *FEAS*-strategy of retiming critical vertices. In order to identify feasible register arrangements in a branch bundle, we have to consider all end vertices of the bundle simultaneously. Furthermore, we have to consider data arrival times at registers positioned on edges of that bundle as well as data arrival times at registers positioned on outgoing edges of end vertices of the bundle. The details are given in the following subsections.

1) Arrival Time Calculation: Recall that our timing model from Section II-A assigns distinct delays to both vertices and edges. Consequently, we can associate different arrival times with a vertex and its outgoing edge. First, for each edge, we determine t_i, t_o, and t_w for the actual situation. The arrival time $t_{ar}(v)$ of a vertex v is calculated from the delay values of the incoming edges $e_i = (u_i, v)$ and the arrival time of the predecessor vertices u_i of v as follows:

$$t_{ar}(v) = t_d(v) + \max \begin{cases} t_o(e_i), & w_r(e_i) > 0 \\ t_{ar}(u_i) + t_w(e_i), & w_r(e_i) = 0 \end{cases}. \tag{11}$$

For an edge $e = (u, v)$, we define an edge arrival time t_{e_ar}:

$$t_{e\_ar}(e) = t_{ar}(u) + \begin{cases} t_i, & w_r(e) > 0 \\ t_w + t_d(v) + t_{tr}(v), & w_r(e) = 0 \end{cases}. \tag{12}$$

The parameter $t_{tr}(u)$ denotes the *time to register* for a vertex u and is calculated as

$$t_{tr}(u) = \max\left(t_i(e_j)\right), \quad ej = (u, vj) \tag{13}$$

using the t_i values for the case that $w_r(e_j) > 0$. In other words, for the case that there is a register on e, t_{e_ar} denotes the signal arrival time at the input of this register. Otherwise, if no register is present, t_{e_ar} denotes the latest arrival time at an assumed (not necessarily yet present) register on an outgoing edge of v. An illustration of this definition is shown in Fig. 6.

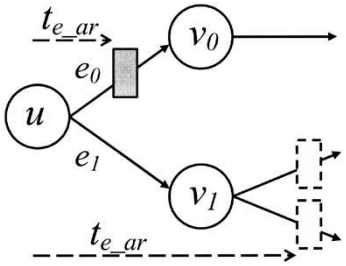

Fig. 6. Definition of edge arrival time.

This calculation model is motivated by the retiming procedure of the following section. This procedure takes into account that a previously critical end vertex v_i of a bundle may become uncritical by retiming another end vertex v_j of the same bundle. In this case, however, since this effect is of limited strength, we assume that an immediate successor of v_i remains critical and needs to be retimed.

2) Retiming Critical Vertices: In contrary to the *FEAS*-algorithm which inspects each vertex separately when deciding whether or not to retime it, we consider all end vertices of a branch bundle at one glance.

This is motivated by the fact that the data arrival times of these vertices depend on each other, and it enables us to detect solutions where retiming some of the end vertices of a bundle can remove a constraint violation in a neighbor branch, as shown in Fig. 7. Retiming gate C allows a register configuration as shown on the right side which leads to a load reduction for gate A. This effect may be large enough to let all critical paths become uncritical.

At the core of our retiming algorithm, we have function *analyze_nets* shown in Fig. 8. It investigates each edge bundle and marks vertices for retiming.

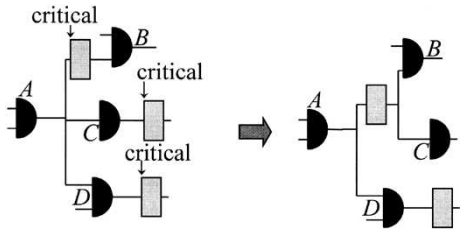

Fig. 7. Retiming example.

```
analyze_nets(t_max) {
  for each branch bundle B = [b_i] = [(u,v_i)]
    if (B is critical) {
      find register arrangements in B reachable by
        backward retiming some vertices v_i that allow
        B to become uncritical;
      if (no arrangement is found)
        mark all successors v_i with w_r(b_i)==0;
      else
        select_retiming();
    }
```

Fig. 8. Algorithm ***analyze_nets(·)***.

Definition 1: An edge b is critical if $t_{e\_ar}(b) > t_{\max}$.

Definition 2: An edge bundle B is critical if at least one edge $b \in B$ is critical.

If an edge bundle $B = \{b_i\} = \{(u, v_i)\}$ is critical, then we try to find a register arrangement on the edges $b \in B$ that is reachable by backward retiming some vertices v_i. If an appropriate arrangement is found, we mark all vertices that need to be retimed in order to achieve this arrangement. If more than one solution is found, function *select_retiming* chooses one of them heuristically using the following heuristic.

1) Choose the solution forcing the minimum number of end vertices of uncritical edges to be retimed.
2) If multiple choices still exist, choose the one forcing a minimum number of end vertices of critical edges to be retimed.
3) If multiple solutions still exist, choose the one minimizing $(t_{\max} - \max(t_{e\_ar}(b))b \in B$.

The idea behind this heuristic is to perform only as much retiming as necessary to remove local timing constraint violations, i.e., we attempt to be as conservative as possible. In this way, we make sure that we do not destroy solutions that may be found in a later processing step. This is also important for our theoretical considerations in Section III. If no arrangement is found, all vertices v_i with $w_r(b_i) = 0$, $b_i = (u, v_i)$ are marked.

The basic philosophy of our procedure is to exploit load coupling between branches in order to extend the number of feasible retimings in a fanout system. By exploring these additional possibilities, our approach may find a feasible retiming where the conventional approach (which considers each vertex separately) fails.

Our modified *FEAS*-algorithm using the coupled edge timing model (*FEAS_CTM*) tries to find a retiming for a given cycle time $t_{\max}$, as shown in Fig. 9.

If no feasible retiming exists, the original *FEAS* algorithm detects this in $|V|$ iterations of its inner loop. Standard *FEAS* tries

```
FEAS_CTM(t_max) {
  for (i = 0; i < |E|, i = i+1)
    calculate_arrivaltimes();
    if (checkcycletime(t_max) == true)
      return true;
    analyze_nets(t_max);
    for each (vertex v)
      if (v is marked)
        r(v) = r(v) + 1;
  return false;
}
```

Fig. 9. Algorithm ***FEAS_CTM(·)***.

to satisfy violated timing constraints locally by retiming a critical vertex u. *FEAS_CTM*, however, first attempts to resolve the violation by retiming one or more successors v_i of u, whereas u itself eventually may be retimed during a later iteration. Therefore, *FEAS_CTM* needs

$$\sum_{u \in V} |\{(u, v_i)\}| = |E|$$

iterations to test whether or not it can reach a feasible retiming.

III. Optimality Issues

An important issue for any retiming algorithm is the cycle time it achieves. The original *FEAS* algorithm guarantees that it will find a solution for a given cycle time if at least one solution exists under the assumption that all vertex delays have a nonnegative value. For VLSI circuits, this is always the case. For the more accurate Soyata–Friedman timing model the authors of [14] proposed two approaches. For the general case where no assumptions are made about edge delays in the retiming graph, they developed a branch and bound based retiming algorithm which guarantees that a solution will be found if one exists. However, this approach suffers from tremendous CPU runtimes for larger circuits. Therefore, it is applicable only for very small circuits. For the case that the monotonicity constraint is not violated they showed that a standard algorithm for solving linear inequalities, e.g., the Bellman–Ford algorithm, also guarantees that a solution will be found if one exists. It has been shown in [2] that a relaxation algorithm like *FEAS* operating directly on a retiming graph performs the same operations as Bellman–Ford would perform on a constraint graph that has been derived from the same retiming graph. For reasons of computational efficiency, the Soyata–Friedman model is of practical interest only in combination with an efficient constraint solver like Bellman–Ford. Therefore, we consider this combination in this section and use it as reference point for the following theoretical analysis of our approach.

Our new timing model allows a more precise representation of a circuit than previous models and, consequently, it allows finding feasible solutions that cannot be detected using simpler models. However, because of the coupling between the edges, our heuristics for selecting vertices for retiming cannot guarantee that local decisions always lead to the global optimum. Hence, compared to standard *FEAS* or the above-mentioned combination of Soyata–Friedman model and the Bellman–Ford

algorithm, we lose an attractive theoretical property. On the other hand, it turns out to be beneficial to sacrifice this theoretical property. Known approaches are only optimal within their inaccurate timing model. Their solutions may not be optimal any more if delays are calculated in a more accurate way, e.g., in the circuit netlist after remapping. Consequently, for comparing the quality of solutions found by various algorithms, it is necessary to consider the accuracy of the timing model being used.

Taking this into account, it is possible to prove that if the Bellman–Ford algorithm using the Soyata–Friedman timing model finds a feasible solution for a particular cycle time, then *FEAS_CTM* always finds one, too. Consequently, in a binary search-based optimization using *FEAS_CTM* results in a smaller or at least the same cycle time than using Bellman–Ford/Soyata–Friedman. This is shown in the following.

Because the algorithms use different timing models, different methods for mapping a circuit netlist onto a retiming graph are required. Consequently, for a comparison of the results achieved by the algorithms, we have to take into account how vertex delays and edge delays in the retiming graph are derived from gate delays and wire delays in the circuit netlist.

For our timing model, this has been described already in the previous sections. In the following, we consider the Soyata–Friedman model. We have already explained that the delay for a signal propagating from one gate to an immediate successor gate depends on the register arrangement in neighbor paths of the fanout system. However, the Soyata–Friedman model does not allow us to assign multiple delay values to an edge as would be necessary for taking these dependencies into consideration, so we have to opt for one particular value. If we chose optimistic values for the edge delays, then the Soyata–Friedman approach would achieve a solution that may not be feasible anymore after mapping the retiming graph back onto a circuit netlist. Consequently, if we want to be sure that the final circuit reaches at least the cycle time that has been measured in the retiming graph, it is mandatory to use pessimistic values for t_i, t_o, and t_w.

In other words, we can consider the Soyata–Friedman model as a simplified version of our model where each delay table contains only one entry. This entry corresponds to the maximum entry in the corresponding table in our model. This leads to the following.

Observation 1: A Soyata–Friedman retiming graph G^* can be derived from a *FEAS_CTM* retiming graph G by replacing each edge delay table by the maximum entry of that table.

For the remainder of the proof, we assume that monotonicity constraints are not violated in G^* because this is a precondition for guaranteeing optimality when using Bellman–Ford.

Definition 3: An edge weighting (e.g., a register placement) in a retiming graph is feasible if each register to register path does not exceed the target cycle time $t_{\max}$.

Lemma 1: For each edge $e \in G$, the delay value(s) holding for any edge weighting in G are less or equal to the corresponding delay values of the corresponding edge $e^* \in G^*$.

Proof: Follows from the way G^* is derived as described by Observation 1.

Lemma 2: For a particular edge weighting, each path in G has a smaller or the same delay than the corresponding path in G^*.

Proof: Results from Lemma 1. If each edge in G has a delay less or equal to the delay of the corresponding edge in G^*, then the same holds for each path.

Lemma 3: If an edge weighting in G^* is feasible, then the same edge weighting in G is also feasible.

Proof: Results from Lemma 2.

Lemma 3 tells us that if a feasible solution is found in G^* by Bellman–Ford, then the same solution is also detected to be feasible in G by *FEAS_CTM*.

Next, we show that if Bellman–Ford finds a feasible edge weighting in G^*, then *FEAS_CTM* finds the same edge weighting in G except in the case when *FEAS_CTM* previously finds another edge weighting which is also feasible. In those cases, *FEAS_CTM* terminates and returns the feasible solution found first. The basis for the proof is given by the behavior of *FEAS_CTM*. *FEAS_CTM* attempts to remove a constraint violation in G always using the same, or a more conservative, retiming than the one Bellman–Ford would apply in G^*. This means that each solution found by Bellman–Ford in G^* is reachable by *FEAS_CTM* in G.

More formally, we can state it as follows.

Theorem 1: If *FEAS_CTM* decides to retime a critical vertex u to remove a constraint violation in G, Bellman–Ford also retimes the corresponding vertex u^* in G^*.

(Note that the opposite does not hold. This is the reason why *FEAS_CTM* is able to detect feasible solutions in G that cannot be detected in G^*.)

Proof: Let us first review under which conditions a vertex $u^* \in G^*$ is retimed. In G^* there are no dependencies between edge delay values and register arrangements in neighbor paths. Vertex u^* is considered to be critical (and is retimed) if the data arrival time at the input of a register located on an outgoing edge exceeds $t_{\max}$, e.g., if

$$t_{\max} < t_{ar}(u^*) + \max\left(t_i(e_j)\right), \quad ej = (u^*, vj^*). \tag{14}$$

Now, let us investigate under which circumstances a vertex is retimed by *FEAS_CTM*. A vertex $u \in G$ may be retimed only if at least one incoming edge of u belongs to a critical edge bundle. Let us denote this edge by e and the corresponding critical bundle by B. We have to distinguish the following cases.

Case 1) e is critical.

Case 1.A: $w_r(e) > 0$: This means that there is already at least one register present on e. Remember that in that case function *analyze_nets* does not move another register onto e, i.e., u is not retimed during this iteration.

Case 1.B: $w_r(e) = 0$: in this case *select_retiming* may select u for retiming. From equations 12 and 13, it follows that

$$t_{\max} < t_{ar}(u) + \max\left(t_i(e_j)\right), \quad ej = (u, vj) \tag{15}$$

holds. From Lemma 1 and Lemma 2 it follows that if 15 holds for $u \in G$, then 14 holds for the corresponding vertex $u^* \in G^*$, too. Consequently, u^* is retimed, too.

Case 2) e is not critical, i.e., some other edge(s) of the bundle is (are) critical: We distinguish further:

Case 2.A: $w_r(e) > 0$: For the same reason as in case 1.A, u is not retimed.

Case 2.B: $w_r(e) = 0$: In this case, u may be retimed despite e is uncritical. We have to distinguish the following further.

Case 2.B.a: Retiming the end vertices only of (some or all) critical edges of B lets B become uncritical.

In this case, u is not retimed. Remember that *analyze_nets* tries to minimize the number of retimed vertices that are end vertices of uncritical edges. (This number is zero in this case).

Case 2.B.b: Considering the end vertices only of critical edges in B for retiming is not sufficient to let B become uncritical i.e., it always leaves some edges in B critical.

In this case, *select_retiming* decides to retime some end vertices of uncritical edges, i.e., u may be retimed. Now, let us consider the corresponding vertex $u^* \in G^*$. If retiming the end vertices only of critical edges in B always leaves some edges in B critical, then applying the same retiming operations to the corresponding vertices in G^* leaves the corresponding edges in G^* critical. Consequently, 0 holds for the start vertex v^* of these edges and, therefore, v^* is retimed. But this means that all successor vertices of v^* being reachable via zero weighted edges are retimed, too. Consequently, u^* is retimed, too.

Theorem 1 tells us that Bellman–Ford moves every register across a larger or at least the same number of vertices as *FEAS_CTM*. Informally spoken, Bellman–Ford is always ahead of *FEAS_CTM* with shifting registers. It remains to show that in those cases when Bellman–Ford finds a feasible solution *FEAS_CTM* also stops at that solution at last, i.e., *FEAS_CTM* does not jump over that solution. In general, *FEAS_CTM* may jump over feasible solutions because we do not presume monotonicity in G. However, because monotonicity is fulfilled in G^*, any solution being feasible in G^* is not skipped by *FEAS_CTM* in G.

An example showing the difference in the behavior of *FEAS_CTM* and Soyata–Friedman/Bellman–Ford is shown in Fig. 10.

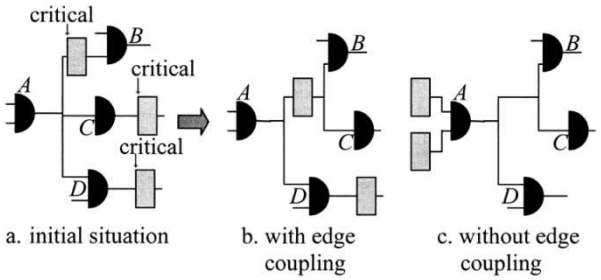

Fig. 10. Retiming with and without edge coupling.

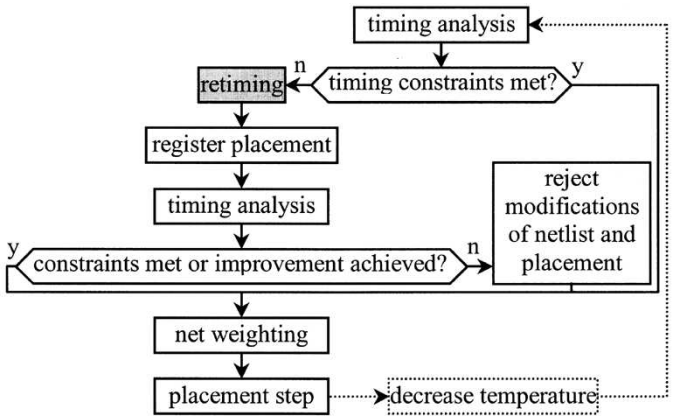

Fig. 11. Placement at a particular temperature level.

IV. Integration Into Placement

A. Overview

The core of our approach is a timing driven simulated annealing-based standard-cell placement algorithm following the philosophy of common placement tools such as [18]. Note that *FEAS_CTM* could also be integrated into other placement algorithms which allow netlist modifications during the placement process. Fig. 11 gives an overview of the placement procedure at a particular temperature level. First, a static timing analysis is performed. For this analysis, wire-length estimations obtained from the actual placement are used. If timing constraints are already met, we continue with the placement process immediately. Theoretically, it would be possible to stop the placement process if timing constraints are already met at the beginning. However, continuing with the placement process in general makes sense because a further reduction of total wire length often can lead to a more compact solution. If timing constraints are not met at this point, a retiming-based optimization step is performed. Afterwards, the newly created registers are inserted into the placement using a fast placement approach. Then, wire lengths are reestimated and the cycle time is calculated again. If constraints are met now, or at least an improvement has been achieved, the new configuration is accepted, otherwise, all modifications of the netlist structure and the placement are to be rejected. Afterwards, net weights are recalculated and the placer begins another iteration.

The following sections describe the steps of the placement process in more detail.

B. Register Placement

In general, a simulated annealing-based placer is able to find good positions for the newly created registers, independent of their initial position. But this process takes a lot of time if registers are inserted randomly, making it impossible to verify immediately after register insertion whether or not a cycle-time improvement has been really achieved.

Furthermore, it saves a lot of work for the placer if new registers are inserted at "reasonable" locations, especially at low temperatures when cells are not allowed to make large jumps.

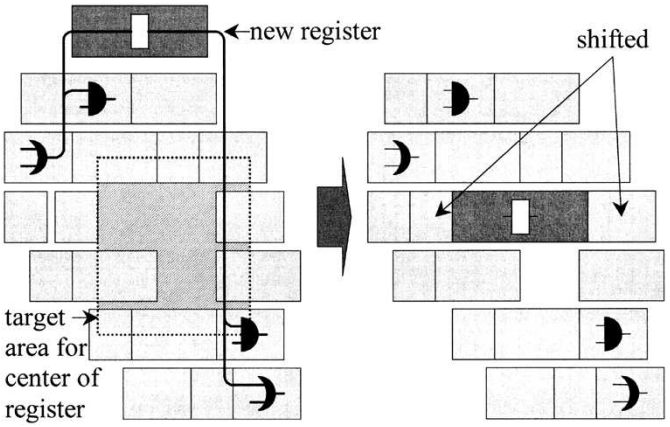

Fig. 12. Single register placement example.

Therefore, we use a separate register insertion step to provide the timing analyzer quickly with realistic assumptions about the wire lengths after retiming has been performed. For each new register, a position is determined such that the sum of the lengths of the nets connected to this register is minimized. In many situations, the result is not a particular vertex, but a target area of rectangular shape. In the latter case, we look for the most suitable cell gap inside this area and position the register there. This helps to keep the modifications of the original placement as small as possible. If the gap is not large enough, neighbor cells are pushed aside first. By doing so, it is always guaranteed that no cell overlapping occurs. At this point, no further work is done to reuse gaps left by deleted registers. No work is done either to balance the total row length because these tasks are performed by the simulated annealing placer later. An example of inserting a single additional register is shown in Fig. 12.

C. Repeated Check of Retiming and Placement

After retiming has been performed and the newly introduced registers have been added to the placement, wire lengths are estimated again and timing analysis is repeated to check whether cycle time really has been improved. At a first glance, this check may seem unnecessary because retiming-based cycle time optimization of a synchronous network should at least not deteriorate the cycle time. However, in our experiments it turned out that, when the effect of retiming on the placement is taken into account, it is indeed possible that a retiming step increases cycle time. The first reason for this is that retiming may increase the number of registers in a circuit, sometimes dramatically. In typical standard cell libraries, flip flops and latches have a far greater area requirement than simple logic cells. Thus, already small increases in register count may result in a significant increase in area requirement. This leads to longer wires and may offset the performance gain achieved by retiming. The second reason is that modifying placement changes the positions of cells and the lengths of nets which are not directly affected by retiming. These effects cannot be taken into account by the retiming algorithm and may also result in cycle-time deterioration.

Therefore, a newly retimed configuration is only accepted if a performance improvement has been achieved immediately after the registers have been inserted. Otherwise, the modifications of placement and netlist are rejected.

TABLE I
CYCLE TIME AND REGISTER COUNTS FOR *FEAS_CTM*

circuit	None	Pre-placement		Post-placement		Tight integration	
	c.t.	c.t.	#FF	c.t.	#FF	c.t.	#FF
S1423	12.4	10.6	113	10.3	111	9.48	111
S1488	4.39	4.15	42	3.83	21	3.25	13
S1494	4.46	3.77	39	4.01	22	3.18	12
S5387	3.95	4.12	319	3.94	164	3.90	164
S9234	10.2	7.29	447	7.24	228	6.40	451
S9234.1	10.3	8.04	425	7.70	211	7.31	438
S13207	10.0	8.42	862	9.81	669	7.39	872
S13207.1	10.5	9.68	640	9.38	641	9.35	641
S15850	15.3	11.3	960	14.5	597	8.21	1088
S15850.1	14.8	12.8	587	11.4	586	10.7	605
S35932	10.6	10.4	1728	10.6	1728	8.34	2659
S38584	17.0	17.6	3205	16.2	1452	14.6	1452
S38584.1	14.1	15.3	3413	13.4	1426	12.9	1428
S38417	15.2	12.1	2213	10.7	2193	10.4	2171

D. Accelerating Retiming

In our experiments, we observed that in those cases where *FEAS_CTM* has been able to identify a feasible retiming it needed only a very small number of iterations ($\ll |E|$) of its inner loop. Consequently, the retiming process may be speeded up by using a technique for detecting unreachable cycle times early. The same observation was made by Shenoy and Rudell [3] for the original *FEAS*-algorithm. In our implementation, we use the acceleration technique described in [20].

V. EXPERIMENTAL RESULTS

In order to evaluate the benefit of our new timing model and the tight integration of placement and retiming, a comparison of three different design flows is of interest.

- A conventional design flow consisting of retiming a logic netlist, followed by timing-driven placement.
- A design flow as in [17] consisting of timing-driven placement, followed by retiming using the delay values calculated from the final placement. After performing a register insertion step as described in Section IV-B, some additional placement steps at very low temperatures are performed to achieve uniform row lengths again.
- A tight integration of retiming and placement as described in this paper.

For our experiments, we mapped the larger circuits of the ISCAS'89 benchmark set onto a 0.18-μm-standard cell library. The results are shown in Table I. Column 2 contains the achieved cycle time for a timing-driven placement approach without any application of retiming. Then, for each of the previously described design flows using retiming, the achieved cycle time (c.t.) in nanoseconds and the final number of registers (#FF) are shown. Columns 3 and 4 contain results for preplacement retiming, columns 5 and 6 contain results for postplacement retiming, and columns 7 and 8 show the results for the approach presented in this paper. The wire length values used for cycle time calculation have been estimated using the half-perimeter bounding-box method commonly used in placement tools.

TABLE II
RESULTS FOR STANDARD *FEAS*

circuit	Pre-placement		Post-placement		tight integration	
	c.t.	#FF	c.t.	#FF	c.t.	#FF
S1423	10.6	113	10.7	112	10.6	114
S1488	4.53	7	4.39	6	4.30	6
S1494	4.53	7	4.46	6	4.45	6
S5387	4.37	325	3.95	179	3.85	348
S9234	7.31	268	7.08	249	6.26	462
S9234.1	7.34	269	7.66	242	7.36	553
S13207	9.31	950	10.0	669	8.30	943
S13207.1	9.79	640	9.38	641	9.18	641
S15850	12.5	962	15.3	597	12.6	3355
S15850.1	13.4	586	11.4	610	10.3	659
S35932	10.4	2826	10.5	2193	8.98	2841
S38584	19.0	3379	17.0	1452	16.2	3330
S38584.1	13.0	1428	13.4	1428	12.8	1429
S38417	12.5	2006	10.7	2193	10.2	2479

TABLE III
RESULTS FOR SOYATA–FRIEDMAN TIMING MODEL

circuit	Pre-placement		Post-placement		tight integration	
	c.t.	#FF	c.t.	#FF	c.t.	#FF
S1423	10.6	113	10.3	111	10.6	116
S1488	4.53	7	4.39	6	4.30	6
S1494	4.53	7	4.46	6	4.45	6
S5387	4.14	317	3.94	164	3.92	167
S9234	7.30	325	7.24	228	6.43	399
S9234.1	7.90	308	7.68	240	7.32	251
S13207	9.66	967	10.0	669	8.10	969
S13207.1	9.75	640	9.38	641	9.35	641
S15850	12.0	965	15.3	587	12.0	1079
S15850.1	12.8	598	11.4	610	10.8	648
S35932	10.4	1728	10.6	1728	8.97	2734
S38584	17.8	3177	17.0	1452	15.8	1728
S38584.1	15.2	3414	13.4	14.28	12.9	1428
S38417	12.2	2135	10.7	2193	10.2	2363

In order to show the impact of out new retiming algorithm *FEAS_CTM*, we repeated these experiments for standard *FEAS* (Table II) and for *FEAS_CTM* in combination with the Soyata–Friedman model as described in Section III (Table III).

The experimental results show the following.

1) Using retiming only before placement achieved the smallest performance improvement of all strategies. In a few cases, cycle time was even larger after placement. If retiming was applied after placement, we achieved somewhat better results, and in no case was there was an increase of cycle time. However, this approach was outperformed by using tight integration as proposed in Section IV. It achieved equal or better results for each benchmark.
2) Using standard *FEAS* or retiming based on the Soyata–Friedman model clearly achieved smaller improvements than *FEAS_CTM*. This confirms that a careful analysis of load changes caused by retiming in fanout systems does indeed contribute to a further reduction of cycle time.

Table IV gives a summarizing overview of the approaches by comparing the achieved improvements in cycle time. We note

TABLE IV
ACHIEVED CYCLE TIME IMPROVEMENTS IN PERCENT

	pre-placement retiming			post-placement retiming			tight integration		
	FEAS	SF	FEAS CTM	FEAS	SF	FEAS CTM	FEAS	SF	FEAS CTM
Min	-11.8	-7.8	-8.5	0	0	0.25	0.02	0.22	1.2
Max	28.7	26.3	28.5	31.2	29.6	29.6	38.6	37.0	46.3
Av.	6.7	8.1	11.0	9.98	10.2	12.2	15.9	16.2	23.7

TABLE V
CPU RUNTIMES FOR RETIMING IN SECONDS

circuit	FEAS CTM	FEAS CTM using SF	FEAS	circuit	FEAS CTM	FEAS CTM using SF	FEAS
S1423	0.076	0.012	0.004	S13207.1	5.321	1.123	0.337
S1488	0.153	0.018	0.007	S15850	5.543	1.232	0.387
S1494	0.149	0.014	0.004	S15850.1	7.234	1.543	0.491
S5387	0.843	0.204	0.055	S35932	6.622	1.344	0.420
S9234	2.643	0.256	0.071	S38584	26.032	4.657	1.387
S9234.1	2.532	0.243	0.073	S38584.1	28.965	5.867	1.498
S13207	3.254	0.894	0.275	S38417	20.988	4.322	1.254

that both the coupled-edge time model and the integration of retiming into placement substantially contribute to the quality of our results.

Table V contains the CPU run times in seconds on a Sun Ultra Sparc 5 Workstation for trying to reach a certain target cycle time using various retiming algorithms. The runtimes are average values being derived from numerous optimization runs using different placements and different target cycle times. The table shows that using a coupled-edge timing model (CTM) instead of the Soyata–Friedman model (abbreviated by SF) enlarges the runtime by a factor of 4–10. However, the total CPU time spent for any of the retiming methods is negligible compared to the computational effort spent in standard placement algorithms or other optimizations performed during a typical design flow.

Overall, we observe that it is indeed of great interest to investigate accurate timing models for retiming as well as the integration of retiming into placement. In comparison with the conventional design flow (preplacement standard *FEAS*, followed by timing driven placement), our new approach (tight coupling of *FEAS_CTM* and placement) achieved an improvement in cycle time of up to 34% and 17% on the average.

VI. CONCLUSION

A new retiming algorithm has been developed using a highly accurate timing model. This allows to model timing at fanout trees correctly. In general, our approach pursues the same basic retiming strategy as the conventional *FEAS* algorithm leading to low complexity of the overall procedure. However, a more detailed local analysis of fanout systems improves the accuracy of the timing data and makes it possible to identify better register locations than in previous approaches. Furthermore, we present an approach for integrating retiming into the physical design process. Instead of using retiming as a preplacement or a postplacement optimization method it is applied as a cycle time improvement technique throughout the whole placement

process. The experimental results show that our integrated approach exploits the optimization potential of retiming and placement significantly better than applying retiming only before or after placement.

REFERENCES

[1] C. Leiserson and B. Saxe, "Optimizing synchronous systems," *J. VLSI Comput. Syst.*, vol. 1, no. 1, pp. 41–67, 1983.

[2] ——, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[3] N. Shenoy and R. Rudell, "Efficient implementation of retiming," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1994, pp. 226–233.

[4] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 74–84, Jan. 1991.

[5] B. Lockyear and C. Ebeling, "Optimal retiming of level-clocked circuits using symmetric clock schedules," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1097–1109, Sept. 1994.

[6] A. Ishii, C. Leiserson, and M. Papaefthymiou, "Optimizing two-phase, level-clocked circuitry," in *Proc. Brown/MIT Conf., Adv. Res. VLSI Parallel Syst.*, 1992, pp. 246–264.

[7] M. Papaefthymiou, "Asymptotically efficient retiming under setup and hold constraints," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1998, pp. 288–295.

[8] V. Sundararajan, S. Sapatnekar, and K. Parhi, "MARSH: min-area retiming with setup and hold constraints," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1999, pp. 2–13.

[9] K. Eckl, J. Madre, P. Zepter, and C. Legl, "A practical approach to multiple-class retiming," in *Proc. ACM/IEEE Design Automation Conf.*, 1999, pp. 237–242.

[10] A. El-Maleh, T. Marchok, J. Rajski, and W. Maly, "Behavior and testability preservation under the retiming transformation," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 528–542, May 1997.

[11] D. Stoffel and W. Kunz, "Record & play: A structural fixed point iteration for sequential circuit verification," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1997, pp. 394–399.

[12] C. van Eijk, "Sequential equivalence checking without state space traversal," in *Proc. Design Automation Test Eur.*, 1998, pp. 618–623.

[13] T. Soyata and E. Friedman, "Retiming with nonzero clock skew, variable register, and interconnect delay," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1994, pp. 234–241.

[14] T. Soyata, E. Friedman, and J. Mulligan, "Incorporating interconnect, register, and clock distribution delays into the retiming process," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 105–120, Jan. 1997.

[15] K. Lalgudi and M. Papaefthymiou, "DELAY: An efficient tool for retiming with realistic delay modeling," in *Proc. ACM/IEEE Design Automation Conf.*, 1995, pp. 304–309.

[16] J. Cong and S. Lim, "Physical planning with retiming," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2000, pp. 2–7.

[17] T. Tien, H. Su, Y. Tsay, Y. Chou, and Y. Lin, "Integrating logic retiming and register placement," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 1998, pp. 136–139.

[18] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*. Norwell, MA: Kluwer, 1988.

[19] I. Neumann and W. Kunz, "Placement driven retiming with a coupled edge timing model," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, 2001, pp. 95–102.

[20] I. Neumann, K. Sulimma, and W. Kunz, "Accelerating retiming under the coupled-edge timing model," in *Proc. IEEE Annu. Symp. VLSI*, Apr. 2002, pp. 135–140.

Ingmar Neumann received the DipI.-Ing. degree in electrical engineering from the University of Karlsruhe, Karlsruhe, Germany, in 1992 and the Ph.D. degree in computer science from the University of Technology of Berlin, Berlin, Germany, in 1998.

From 1998 to 2001, he held a Postdoctoral position in the Computer Science Department at the University of Frankfurt a.M., Frankfurt, Germany. Currently, he is with the Electronic Design Automation Group of the Department of Electrical Engineering at the University of Kaiserslautern, Kaiserslautern, Germany. His research interests include VLSI logic and layout synthesis, timing analysis, and layout-driven logic optimization for deep-submicron technologies.

Wolfgang Kunz (S'90–M'91) received the Dipl.Ing. degree in electrical engineering from the University of Karlsruhe, Karlsruhe, Germany, in 1989 and the Ph.D. degree from the University of Hannover, Hannover, Germany.

From 1989 to 1991, he was with the Electrical and Computer Engineering Department at the University of Massachusetts, Amherst. From 1993 to 1998, he was with Max Planck Fault-Tolerant Computing Group at the University of Potsdam. From 1998 to 2001, he was a Professor in the Department of Computer Science at the University of Frankfurt. Since 2001, he has been a Professor with the Department of Electrical Engineering at the University of Kaiserslautern, Kaiserslautern, Germany. He conducts research in the areas of logic and layout synthesis, test generation, and formal hardware verification.

Prof. Kunz has received several awards including the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN Best Paper Award. He is a member of the Association for Computing Machinery.