CLOCK SCHEDULING AND CLOCKTREE CONSTRUCTION FOR HIGH PERFORMANCE ASICS

Stephan Held, Bernhard Korte, Jens Maßberg, Matthias Ringe[⋆]*, Jens Vygen*

Research Institute for Discrete Mathematics, University of Bonn
[⋆]IBM Deutschland Entwicklung GmbH
Lennéstr. 2, 53113 Bonn, Germany

ABSTRACT

In this paper we present a new method for clock scheduling and clocktree construction that improves the performance of high-end ASICs significantly.

First, we compute a clock schedule that yields the optimum cycle time and the best possible clock distribution with respect to early and late mode; in particular the number of critical tests is minimized. Second, individual arrival time intervals are computed for all endpoints of the clocktree. Finally, we construct a clocktree that realizes arrival times within these intervals and exploits positive slacks to save power consumption.

We demonstrate the superiority of our method to previous approaches by experimental results on industrial ASICs with up to 194 000 registers and more than 160 clock domains. We improved the clock frequencies by 5-28% up to 1.033 GHz (in hardware).

1. INTRODUCTION

Due to increase of clock frequencies and design sizes clocktree construction has become one of the most challenging problems in VLSI-design. There is an extensive literature. Most works either focus on clock scheduling or on clocktree construction without optimizing schedules.

A linear programming formulation for cycle time optimization by clock scheduling can be found [9, 16]. Shortest path approaches from graph theory were proposed in [6, 17, 18]. Scheduling techniques that optimize not only the cycle time but also less critical cycles were developed in [1, 14].

Most papers on clocktree construction aim at zero-skew trees or bounded-skew trees and are based on the deferred-merge-embedding algorithm [3, 5, 8, 20]. Methods for integrated buffer/inverter insertion are given in [4, chapter 5], which also gives an overview on the topic.

Kourtev and Friedman proposed a simultaneous clock scheduling and clocktree synthesis [12]. In [13] they considered a quadratic programming approach to increase reliability of the clock schedule. An overview on their work can be found in [14].

Individual skew bounds for each register pair are considered in [15, 19, 22].

All previous works concerning clock scheduling used a register graph, consisting of all registers as nodes and having arcs representing the minimum and maximum delay of data paths between these registers. However, the size of the register graph, with a potentially quadratic number of arcs, proves impracticable for designs with several hundred thousand registers.

Instead, we apply clock scheduling based on a slack balance graph that is based on the timing graph. The size of the slack balance graph is proportional to the size of the netlist, which is typically linear in the number of registers. In particular we can handle very complex chips with many clock domains and complicated timing assertions.

We then apply the minimum balance algorithm [1, 23]. Our clock schedule computation optimizes slacks below a threshold and additionally distributes remaining positive slacks to required clock arrival time intervals. These intervals are used to construct low-power clocktrees.

Clocktree construction is done by a bottom-up algorithm. We obtain the necessary skews by varying the number of stages and the inverter types but not by routing detours. The placement of the inverters (avoiding blockages) and the choice of the inverter sizes is done late in the algorithm to save power consumption.

Our experimental results prove that this approach is feasible even for the most complex ASICs that are designed today. We improve clock frequencies and power consumption simultaneously.

2. STATIC TIMING ANALYSIS

We will now give a short introduction to our static timing analysis model. This model was basically given in [11]. We have a finite set $\mathcal{M}$ of timing measurement points on our chip. In general these points correspond to the pins but there can also be other, artificial or user defined timing points.

Starting at primary inputs and at outputs of signal-triggering registers we compute delays between consecutive measurement points and store four arrival times at all points along the combinational logic: That are the latest rising (AT_r), the earliest rising (at_r), the latest falling (AT_f), and the earliest falling (at_f) arrival times of a signal. At the end of the combinatorial paths we test whether the signals arrive in time. Signals of different origins are merged by retaining only the most critical signals.

As delays depend significantly on signal slews, we also propagate slew values to compute timing. Using enhanced slew propagation models [2, 21], the effect of strong delay variations due to discontinuity is weakened. For simplicity slews are omitted here.

As stated before we have a large number of different clock domains with different frequencies on current ASIC chips. A clock domain identifier is stored with the signals. Signals of different domains are not merged, but there may be timing constraints between signals from different domains. For simplicity we restrict our description to the case of a single clock domain. The formulation can be generalized easily to multiple clock domains. We now describe how signals are merged and tests are performed.

2.1. Propagation Constraints

At primary inputs $d_{pi} \in \mathcal{M}$ we are given earliest and latest start times: $\mathrm{st}_x(d_{pi})$ and $\mathrm{ST}_x(d_{pi})$ ($x \in \{r, f\}$). The arrival times used in optimization must fulfill

$$\mathrm{AT}_x(d_{pi}) \geq ST_x(d_{pi}), \quad x \in \{r, f\} \quad \text{and} \tag{1}$$
$$\mathrm{at}_x(d_{pi}) \leq st_x(d_{pi}), \quad x \in \{r, f\}. \tag{2}$$

To be least pessimistic the inequalities are usually set to equality. Let $a \in \mathcal{M}$ and $b \in \mathcal{M}$ be two consecutive measurement points, where a signal $x \in \{r, f\}$ in a directly causes a signal $y \in \{r, f\}$ in b. As the latest/earliest signal arrival time in b is the maximum/minimum over all latest/earliest signals propagated from points preceding b, we have following inequalities:

$$\mathrm{AT}_x(a) + \mathrm{DELAY}(a, b, x, y) \leq \mathrm{AT}_y(b) \quad \text{and} \tag{3}$$
$$\mathrm{at}_x(a) + \mathrm{delay}(a, b, x, y) \geq \mathrm{at}_y(b), \tag{4}$$

where $x, y \in \{r, f\}$ and DELAY and delay are the maximum and minimum delay. Constraints of type (1)–(4) are called propagation constraints, as the merging of signals during arrival time propagation is based on them.

2.2. Test Constraints

At the end of logic paths there are tests whether signals arrive in time. For each signal – described by $x \in \{r, f\}$ – reaching a primary output $d_{po} \in \mathcal{M}$ there is an earliest and a latest required arrival time defined, named $\mathrm{rat}_x(d_{po})$ and $\mathrm{RAT}_x(d_{po})$. For correct functioning of the chip, following inequalities must be guaranteed:

$$\mathrm{AT}_x(d_{po}) \leq \mathrm{RAT}_x(d_{po}) \quad \text{and} \tag{5}$$
$$\mathrm{at}_x(d_{po}) \geq \mathrm{rat}_x(d_{po}), \tag{6}$$

where $x \in \{r, f\}$. The differences between computed and required arrival times – called **slacks** – are defined as follows:

$$\mathrm{SLACK}_x(d_{po}) := \mathrm{RAT}_x(d_{po}) - \mathrm{AT}_x(d_{po}) \quad \text{and} \tag{7}$$
$$\mathrm{slack}_x(d_{po}) := \mathrm{at}_x(d_{po}) - \mathrm{rat}_x(d_{po}), \tag{8}$$

$x \in \{r, f\}$.

Signals ending at registers are tested against clock signals at the clock pins. First, there are **setup tests**, where data signals arriving at the input pin d_{in} are tested to arrive a small time (setup-time) before the earliest possible latch closing time in the next cycle $at_{cl}(d_{clock}) + \mathrm{adjust}$. The value adjust is usually the cycle time, but can differ. Generally a latest arrival time is required to be earlier than an earliest arrival time.

$$\mathrm{AT}_x(d_{in}) + \mathrm{setup}_x \leq at_{cl}(d_{clock}) + \mathrm{adjust}, \tag{9}$$

$x \in \{r, f\}$, where $cl \in \{r, f\}$ denotes the closing edge of the latch. The setup test induces a required arrival time and a late slack for the data phase.

$$\mathrm{RAT}_x(d_{in}) := \mathrm{at}_{cl}(d_{clock}) + \mathrm{adjust} - \mathrm{setup}_x, \tag{10}$$
$$\mathrm{SLACK}_x(d_{in}) := \mathrm{RAT}_x(d_{in}) - \mathrm{AT}_x(d_{in}) \tag{11}$$

$x \in \{r, f\}$. Second, there are **hold tests**, which determine whether signals arrive too early. Earliest arrival times of data signals are tested against the latest time that the latch could close in the same cycle:

$$at_x(d_{in}) \geq \mathrm{AT}_{cl}(d_{clock}) + \mathrm{hold}_x, \quad x \in \{r, f\}. \tag{12}$$

The earliest required arrival time and an early slack is defined by

$$\mathrm{rat}_x(d_{in}) := \mathrm{AT}_{cl}(d_{clock}) + \mathrm{hold}_x, \tag{13}$$
$$\mathrm{slack}_x(d_{in}) := \mathrm{at}_x(d_{in}) - \mathrm{rat}_x(d_{in}), \tag{14}$$

$x \in \{r, f\}$. A chip works correctly if all inequalities are met, or equivalently all slacks are positive. All direct inequalities between arrival times on a chip have a common structure:

$$\alpha + c(\alpha, \beta) \geq \beta. \tag{15}$$

where α, β are arrival times and $c(\alpha, \beta)$ is some sum of delay, setup time, adjust, etc. Many further timing constraints can be expressed using such inequalities. An inequality-system of type (15) is well known from admissible potentials in shortest path theory.

3. COMPUTING AN OPTIMUM CLOCK SCHEDULE

Our main goal is to maximize the worst slack. The next goal is to maximize the second smallest slack, and so on. More precisely, we look for arrival times such that the vector of slacks, after sorting in nondecreasing order, is lexicographically maximum. This problem can be solved by applying the minimum balance algorithm [1, 23] to the so-called slack balance graph, to be defined now.

3.1. The Slack Balance Graph

The slack balance graph is a digraph G with weights $c : E(G) \to \mathbb{R}$. It is defined as follows. Each earliest/latest, rising/falling (transition) edge at a measurement point is represented by a node in $V(G)$. Additionally $V(G)$ contains an extra node $\tilde{v}$ that represents the time origin (zero time). Every arrival time can now be represented by a node potential. We introduce an arc from node $v \in V(G)$ to $w \in V(G)$ if there is some constraint of type (15) between the represented arrival times. Constraints between arrival times and constant values can be represented by arcs that are incident to $\tilde{v}$. Costs are chosen according to (15): A constraint $\alpha + c(\alpha, \beta) \geq \beta$ corresponds to an arc (α, β) with weight $c(\alpha, \beta)$.

We omit all parts of the graph representing clocktree parts that precede scheduled register elements. The arising clock nodes with zero indegree represent the clocktree endpoints.

The clock arrival times of clocktree endpoints are forced to be within user-defined time intervals by arcs that represent inequalities of type (1), (2), (5) and (6). These feasible interval limits can be used to limit the spreading of the computed arrival time intervals. Usually their size is 30% – 80% of the cycle-time.

Before an optimum clock schedule of a chip can be computed in (G, c), some intrinsic conditions for the clock signals must be added. These conditions are given indirectly by the asserted clock arrival times. Let $c \in \mathcal{M}$ be a measurement point of a clock signal at a clocktree endpoint.

First we have to ensure the pulse width of the clock signals; that is the difference between rising and falling edge:

$$\mathrm{at}_r(c) + \mathrm{pulse\_width}(r, f) = \mathrm{at}_f(c) \tag{16}$$

The same equation must hold for latest arrival times.

Second, we have to ensure that earliest clock arrival times are smaller than the corresponding latest arrival times.

$$\mathrm{AT}_x(c) \geq \mathrm{at}_x(c), \quad x \in \{r, f\}. \tag{17}$$

The conditions (16) and (17) can be expressed by weighted arcs as above.

To anticipate variations along clocktree paths, e.g. due to manufacturing tolerances, we can add a positive delay constraint to the right hand side of (17).

3.2. Clock Scheduling

Our aim is to compute arrival times that satisfy all timing constraints and leave as much slack as possible at critical tests. Let $p : E(G) \to \{0, 1\}$ be an indicator function such that we want to distribute slack to all arcs e with $p(e) = 1$. Then we consider the following problem:

MINIMUM BALANCE PROBLEM

Input: A strongly connected weighted digraph (G, c) and a parameter function $p : E(G) \to \{0, 1\}$. $c(W) \geq 0$ for all cycles $W \subset G$ with $p(W) = 0$

Output: A node-potential $\pi : V(E) \to \mathbb{R}$ such that the vector of slacks

$$(\pi(v) - \pi(w) + c(e))_{\substack{e=(v,w)\in E \\ p(e)=1}}$$

is lexicographically maximum after sorting the entries in nondecreasing order.

[1] shows how to solve this problem by the minimum balance algorithm – originally developed by [23] for the special case $p \equiv 1$. The worst-case running time is $O(nm + n^2 \log n)$, where $n := |V(G)|$ and $m := |E(G)|$. The case $p : E(G) \to \mathbb{R}_{\geq 0}$ and other generalizations of the problem are discussed in [10].

Iteratively the minimum ratio cycle W, where $p(W) > 0$ and $c(W)/p(W)$ is minimum, is identified. This cycle is contracted and the algorithm continues on the remaining graph. For details see [1]. The algorithm can be stopped after any minimum ratio cycle computaion. As a result, all slacks up to the value of the last minimum ratio cycle are lexicographically maximum after sorting in nondecreasing order.

3.2.1. Scheduling Data Slacks

As we build up the slack balance graph, arcs representing test inequalities are parameterized. Flushing arcs of level-sensitive latches are parameterized too. Now all cycles either contain a complete data path or result from combinations of arcs defining the feasible intervals for clock arrival times, (16) or (17). In the first case they contain at least one test arc and are therefore parameterized. In the second case the cycle must have positive weight.

As tests have different importance we do not parameterize all test arcs at once. Early mode problems (violations of (2), (4), (6) and (12)) are usually eliminated after clocktree construction because the delay variations of the clocktree must be known to identify the problems. Moreover, early mode problems can be eliminated easily by inserting delay buffers.

Therefore we first optimize the late mode slacks (7) and (11) by neglecting early mode test arcs (6) and (12) and subgraphs that are not in the strongly connected component of G that contains the late tests (5) and (9). Late mode balancing is performed up to a late mode threshold ($\mathrm{ltarget}$). Then the achieved late mode slacks are fixed by subtracting the minimum of the arc slack and $\mathrm{ltarget}$ from the costs for all parameterized arcs and eliminating the parameterization.

Second, we add the neglected early mode arcs and balance up to an early threshold ($\mathrm{etarget}$). Again the achieved slack is fixed. This reduces the number of buffers that need to be inserted in early-mode padding drastically, while not affecting late mode slacks significantly.

3.2.2. Arrival Time Intervals

Finally, after balancing data slacks, we compute clock arrival time intervals for the clocktree endpoints that preserve all data slacks below the corresponding $\mathrm{ltarget}$ or $\mathrm{etarget}$.

Computing intervals is equivalent to the distribution of slacks to the arcs separating early and late clock arrival times (17). The problem can again be solved by the minimum balance algorithm after fixing early mode slacks and parameterizing arcs of type (17).

As only small parts of a chip are really critical we usually reach clock intervals of satisfactory size after a few thousand cycle contractions. Again we use an threshold ($\mathrm{ctarget}$) to stop the balancing process. The complete clock scheduling algorithm runs in three steps:

Algorithm 1: Clock Scheduling

① **Balance late mode slacks**. Tests arcs representing late slacks are parameterized. All other test arcs are ignored. Late slacks are balanced up to $\mathrm{ltarget}$.

② **Balance early mode slacks.** All late slacks are fixed. Early test arcs are added. Early mode slacks are balanced up to $\mathrm{etarget}$.

③ **Compute Clock Intervals.** All data tests arcs are fixed and we parameterize the arcs given by (17). Clock intervals are computed up to $\mathrm{ctarget}$.

Theorem 3.1. *The clock scheduling algorithm has a worst-case running time of $O(nm + n^2 \log n)$. The solution is optimum in the following sense:*

Among all possible clock schedules the vector v_{lt} of late mode slacks below $\mathrm{ltarget}$ *is lexicographically maximum after sorting in nondecreasing order.*

Preserving v_{lt}, the vector v_{et} of all early mode slacks below $\mathrm{etarget}$ *is lexicographically maximum after sorting in nondecreasing order.*

Preserving v_{lt} and v_{et}, the vector v_{ct} of the clock interval sizes below $\mathrm{ctarget}$ *is lexicographically maximum after sorting in nondecreasing order.*

Proof. The theorem follows from the preceding descriptions. □

In practice we observe reasonable running times as the balancing processes are bounded by the thresholds (see table 1). There is a tradeoff between quality of result and running time. However, the worst slacks are always optimum.

4. CLOCKTREE DESIGN

Up to now we have shown how to compute optimal arrival time intervals for the clocktree endpoints. In this section we will present a method to build a clocktree that can realize these intervals.

In addition to the sinks provided with arrival time interval, position in the plane and required parity, we are given a source with its position, starttime, startparity and startslew. As repower circuits we allow a set of inverters $inverterset$ with timing rules and bounds on the capacitance they can drive. Moreover, we have a set of rectangular blockages $blockageset$, a slew bound $maxslew$, and a maximum distance between two adjacent vertices (pins in the same net) $maxdist$, in order to bound wire delay.

The task is now to build an arborescence $G = (V, E)$, where the leaves are the sinks, the root is the source, and the internal vertices are inverters in $inverterset$. Moreover, the internal vertices have to be placed whithin the chip area outside the blockages, and the capacitance limits of the inverters, the slew limit ($maxslew$) and the distance limit ($maxdist$) must be kept. We have to ensure that a signal starting at the source with the given starttime, startslew and startparity reaches each sink within its time interval and with the required parity. The secondary goal is to minimize power consumption.

We will generate the clocktree in a bottom-up manner. The vertices will not be placed until they have got a predecessor. For each vertex we determine an area where we can place it, and an area where we can possibly place its predecessor. All areas which appear in the algorithm will be finite unions of octagons, whose angles are multiples of 45°. By storing an octagon as the intersection of two rectangles, one axis-parallel and one rotated by 45°, all necessary operations (e.g. intersection of two areas, etc.) can be performed efficiently.

The assignment of the internal vertices to the inverters is fixed at the very end, after the whole tree has been built. While constructing the vertices we will compute a set of 'solution candidates' for each newly generated vertex. Each solution candidate consists of an inverter for the vertex and a pointer to a solution candidate of each successor. Using these candidates, a time interval for each internal vertex is computed. The time intervals of the sinks are the given required arrival time intervals.

We call a vertex *active* as long as it does not have a predecessor.

4.1. Main Algorithm

At the beginning, the clocktree just consists of the sinks as leaves of the tree with their corresponding time intervals and required parities. They are all active. The active vertices will be stored in two sets, according to their expected parity.

In each iteration of the main loop a new vertex is generated. The successors of the new vertex are computed using a greedy clustering strategy. The first successor is an active vertex with maximum lower bound of its arrival time interval. In section 4.2 the choice of the other successors will be described.

In the next step the position of the successors, which have become inactive now, are fixed. Then we compute the areas where the new vertex and its predecessor can be placed (section 4.3).

Finally we have to specify a set of solution candidates for the new vertex (section 4.4).

The algorithm terminates the main loop when all remaining active vertices can be connected to the source. Now the topology of the clocktree and its embedding are determined. In the last iteration a set of solution candidates for the source is computed. We choose the solution candidate that has minimum power consumption among those yielding the best realization of the starttime and assign the inverters according to this solution.

4.2. Clustering

The clustering determines the successors of a new vertex.

A cluster of successors for a new vertex is feasible if the following conditions hold:

1. the new vertex can be placed so that its distance to each successor is at most $maxdist$,
2. the intersection of the time intervals of all successors is not empty,
3. all successors expect the same parity, and
4. there exists an inverter that can drive all successors and the net connecting them.

We interpret each vertex as a 3-dimensional object, where the area, in which the predecessor can be placed, lies in the x-y-plane, and the time interval lies on the z-axis.

The new cluster is the 3-dimensional intersection of all successors. If and only if this intersection is nonempty, conditions 1 and 2 hold.

Now we use a greedy strategy for clustering. At the beginning the cluster consists only of one successor: the active vertex with maximum lower bound of its arrival time interval.

In each subsequent step, we look for an active vertex v expecting the same parity, such that the volume of the intersection of the cluster and v is nonempty and maximum among all choices of v. If there is no such vertex, we close this cluster. Otherwise we check if there exists an inverter that can drive the cluster and v. If so, we add v to the cluster, otherwise we will not look at v in this clustering again. Then we continue with the next active vertex v of the right parity.

4.3. Placement of the Vertices

If a vertex v is a leaf of the clocktree, it is already placed. Otherwise v has a set of successors. For each successor there is an admissible area for its predecessor. The vertex v can be placed in the intersection of these areas.

Therefore, the main problem is to compute the area where we can place the predecessor of a vertex. For the placement of a vertex the following conditions have to be kept:

1. the distance between two vertices that are immediate neighbours within the clocktree must not exceed $maxdist$,
2. the vertices must not be placed on a blockage, and
3. the vertices have to be 'near' to the source if the time difference between the starttime at the source and the time interval of the vertex is 'small'.

The last point guarantees that the vertices can be connected to the source when their time intervals reach the starttime of the clock signal. So the main loop will terminate.

Now we will explain in detail how condition 3 is taken care of. We construct a special distance graph by partitioning the chip area by a uniform grid with row and column width $w = \frac{maxdist}{2+a}$, where $a \in \mathbb{N}$ is an approximation factor. If the L_1-distance between the midpoints of two grid squares v_1 and v_2 not completely covered by blockages is $k \cdot w$ with $k \leq a$, then an edge (v_1, v_2) of weight k ($\in \mathbb{N}$) will be inserted. Thus, higher values for a will provide a better approximation of the blockages but will also increase the runtime of the algorithm.

Let $v(p)$ denote the grid square containing a point p. We can then compute the distance $d(v)$ in this graph from each vertex v to $v(s)$, where s is the source, using Dijkstra's algorithm [7] (see Fig. 1).

The following can be shown easily:

Theorem 4.1. *For each point p with $0 < d(v(p)) < \infty$ there exists a set of points $p_0, p_1, \ldots, p_{n-1}, p_n \in \mathbb{R}^2 \setminus blockageset$ where p_0 is the source and $p_n = p$, so that $d(v(p_i)) < d(v(p_{i+1}))$, $\|p_i, p_{i+1}\|_1 \leq maxdist$ for all $i \in \{0, 2, \ldots, n-1\}$ and $n \leq \left\lfloor \frac{label(p)}{a+1} \right\rfloor + \left\lceil \frac{label(p)}{a+1} \right\rceil$.*

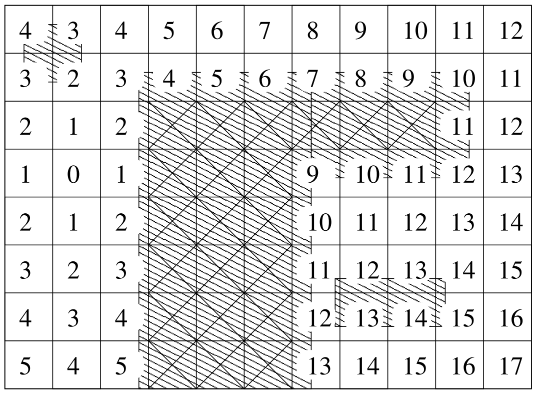

Figure 1: A grid showing the blockages with appropiate labeling ($a = 2$). The source is in the square marked with 0.

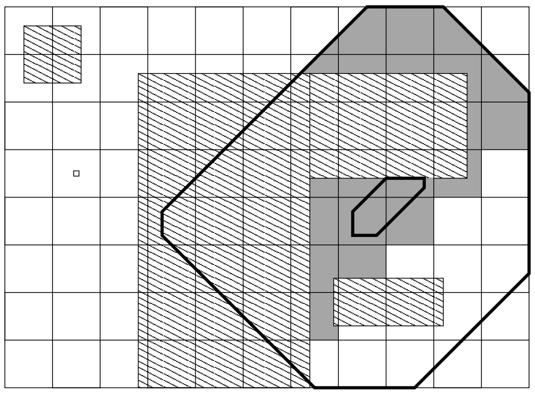

Figure 2: To compute the area where to place the predecessor of a vertex v, we take the placement area of v (small octagon), extend this area by $maxdist$ (big octagon) and then remove the blockages and all nonreachable grid squares (here: with distance > 12). The result is the solid gray area.

For a vertex v we can now compute the area where we can place its predecessor. First we compute the set of all points which have distance at most $maxdist$ to a point where we can place v. From this area we remove the points lying on a blockage. In the last step we have to ensure that condition 3 holds.

Initially we compute the minimum time a signal needs for one length unit. This yields the maximum distance that can be covered until reaching the starttime. We remove all points from our area which have a higher distance label (see Fig. 2).

4.4. Computation of Solution Candidates

A solution candidate of a vertex v is an assignment of all vertices (except leaves) of the subtree rooted at v to the inverter set. Moreover, we associate a slew and a time interval with each solution candidate. If the subtree is realized according to the assignment and a signal starts at v within the time interval and with the given slew, then all sinks in the subtree are reached in time. To bound the number of solution candidates, we discretize the range of possible slews to approximately 20 values.

A solution candidate for a vertex v consists of an inverter i for v and solution candidates for each immediate successor of v. The time interval for the solution candidate is the intersection of the time intervals of the successors plus an offset due to the delay of i. This interval may have negative length, meaning that there are two successors whose intervals do not intersect, and there will be at least one leaf in the subtree that cannot be reached in time. The slew of a solution candidate is computed by backward propagation, assuming that all successors have identical slews.

Each time a new vertex is created, a set of solution candidates is computed for this vertex.

The number of possible inverter assignments for the vertices of a subtree is exponential in their number, and most of the combinations have bad (negative length) time intervals we are not interested in. So we restrict ourselves to solution candidates with maximal time intervals.

We can easily produce such maximal candidates for v from the maximal candidates of the successors (with identical slews) using a sweepline method. To this end, we sort the maximal candidates for each successor by their time intervals (unique because of their maximality). Then we move a sweepline step by step to the lower bounds of the time intervals of all successor candidates (Fig 3). In each step we choose for each successor the latest candidate whose lower bound of the time interval is less or equal to the value of the sweepline. These candidates and a sufficiently strong inverter for v give us a new maximal solution candidate for v.

The number of maximal solution candidates for a vertex v is bounded by

$$(\#\text{leaves in subtree})(\#\text{slews})|inverterset|^{\text{depth of subtree}}.$$

In practice we have a few thousand solution candidates at each vertex.

4.5. Additional Remarks

The described algorithm can easily be extended in order to handle special circuits (e.g. gating circuits or clocksplitters).

In the main algorithm we have two sets S_0 and S_1 of active vertices, each for one parity. Using special circuits we get additional sets describing the logical structure of the clocktree. If a vertex v expects special circuits of the types $T_1, T_2, \ldots, T_n$ (in this order), v will be inserted into an additional set $S_{p,T_n,\ldots,T_1}$, where p is the required parity of v.

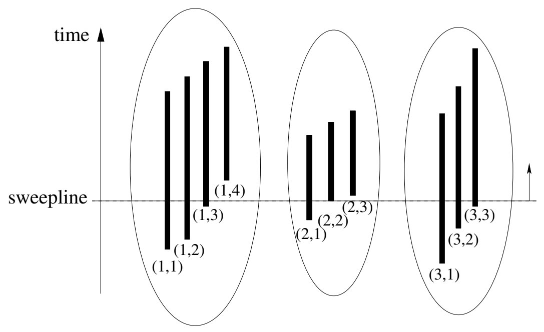

Figure 3: Computation of a maximum solution candidate for a vertex. The sweepline moves upwards, stopping at the lower bound of each interval. In the eighth step shown, candidates $(1,3)$, $(2,2)$ and $(3,3)$ are chosen.

Now the clocktree construction can be done as described above: If a new vertex is created, we have to ensure that all successors belong to the same set, and the new vertex is inserted into its corresponding set.

5. EXPERIMENTAL RESULTS

The proposed clock scheduling and clocktree construction algorithms have been implemented in C and tested on four recent ASICs from industry named 'Jens', 'Matthias', 'Katrin' and 'Alex'.

Two of the chips, 'Matthias' and 'Katrin', have been produced using clocktrees designed by two different industrial zero-skew tools. We compare our results to these trees. 'Jens' is a logic core. 'Alex' is the first ASIC that has been designed using exclusively this approach. We will focus on this chip later.

The timing of the chips was verified by EinsTimer, IBM's static timing analysis tool.

Table 1 (first part) shows the technology, image size, number of circuits, pins and nets, number of clock domains, clocktree sinks, sinks in the biggest clock domain, and the number of edges in the slack balance graph for each chip.

The second part of the table gives the main parameters we used for our algorithm. For better comparison, 'Matthias' and 'Katrin' were tested with slew bounds that result in the same average slews as for the zero-skew trees. A second run was done for 'Katrin' using a smaller slew bound (400ps). On 'Alex' critical clocks were routed on higher layers. Nets on these layers have a lower wire delay so that $maxdist$ can be increased. Moreover, different $maxslew$ values were used for the clock domains on 'Alex' depending on their frequency.

The third part contains the timing results. The fourth line shows the worst slack we reached after running the clock scheduling algorithm. These slacks, computed before constructing the clocktrees, take 300ps for variations in clock paths into account. Without this the slacks would be 300ps greater. The fifth line shows the worst slack after constructing the clocktrees. This is after wiring, 3D extraction and a full timing analysis including on-chip variation. Comparing these values it can be seen that our clocktrees nearly reach the optimal slack. Only the results in the first run on 'Katrin' are slightly worse because of the high slew limit. On 'Jens' and 'Alex' we got even better results, because the clock variation between the sinks on the critical paths was better than assumed. The next line shows the worst slack of the chips with zero-skew trees. Last the resulting frequencies for both approaches are listed. We could increase the maximum frequency of the fastest clock domains by up to 28.4%.

On 'Alex', the worst slack does not occur in the fastest clock domain, therefore the worst slack of this domain is shown in brackets. For the 900MHz domain we reached the design target, while a zero/bounded skew strategy would fail. Moreover, in hardware this domain still runs correctly with a frequency of 1033MHz.

Part 4 of table 1 shows the worst slew, average slew, area consumption by wiring and power consumption. The area consumption by wiring includes the area of each wire and half the minimum distance to an adjacent wire on each side. Thus we get a fair comparison of the consumed routing resources.

The slew bounds were nearly kept. Only on 'Alex' we got some worse slews due to placement legalization after constructing the clocktrees. But only 0.14% of all nets missed the slew bound and less than 0.02% missed it by more than 10%.

For 'Matthias' and 'Katrin' we could compare the power consumption and the area consumption of the zero-skew trees and our trees. On 'Matthias' and in the first run on 'Katrin' we could reduce the power consumption significantly. This is because we did not have to meet time points but time intervals, so we saved balancing resources. Moreover we could reduce the routing resources by using different inverter types instead of routing detours for balancing. In the second test on 'Katrin' we got a higher power consumption: because of the lower slew bound we need more circuits and wiring within the clocktree but also get a smaller cycle time. So we have a tradeoff between lower power consumption and use of routing resources on the one side and higher frequency and a lower slew bound on the other side. The different power consumptions among the chips follow from the different number of sinks, technologies, voltages and frequencies.

The runtime results (part five of table 1; hh:mm:ss) have been obtained on an IBM pSeries 680 Server with a 600 MHz CPU.

Figure 4 shows one clocktree of 'Matthias'.

In the future we will incorporate on-chip variation and tracking into our algorithm more directly. This, however, will require a more advanced timing model. Moreover, there is still room for improving the power consumption. Finally, we plan to integrate placement legalization into the clocktree construction. Nevertheless, we have demonstrated that our method is a big step forward in ASIC clock methodology.

6. REFERENCES

[1] C. Albrecht, B. Korte, J. Schietke, and J. Vygen. Cycle time and slack optimization for VLSI-Chips. *Proc. ICCAD*, 232–238, 1999.

[2] D. Blaauw, V. Zolotov, S. Sundareswaran, C. Oh, and R. Panda. Slope propagation in static timing analysis. *Proc. ICCAD*, 338–343, 2000.

[3] T.-H. Chao, Y.-C. Hsu, and J.M. Ho. Zero skew clock net routing. *Proc. DAC*, 518–523, 1992.

[4] J. Cong, L. He, C.-K. Koh, and P.H. Madden. Performance optimization of vlsi interconnect layout. *Integration, the VLSI Journal 21*, 1–94, 1996.

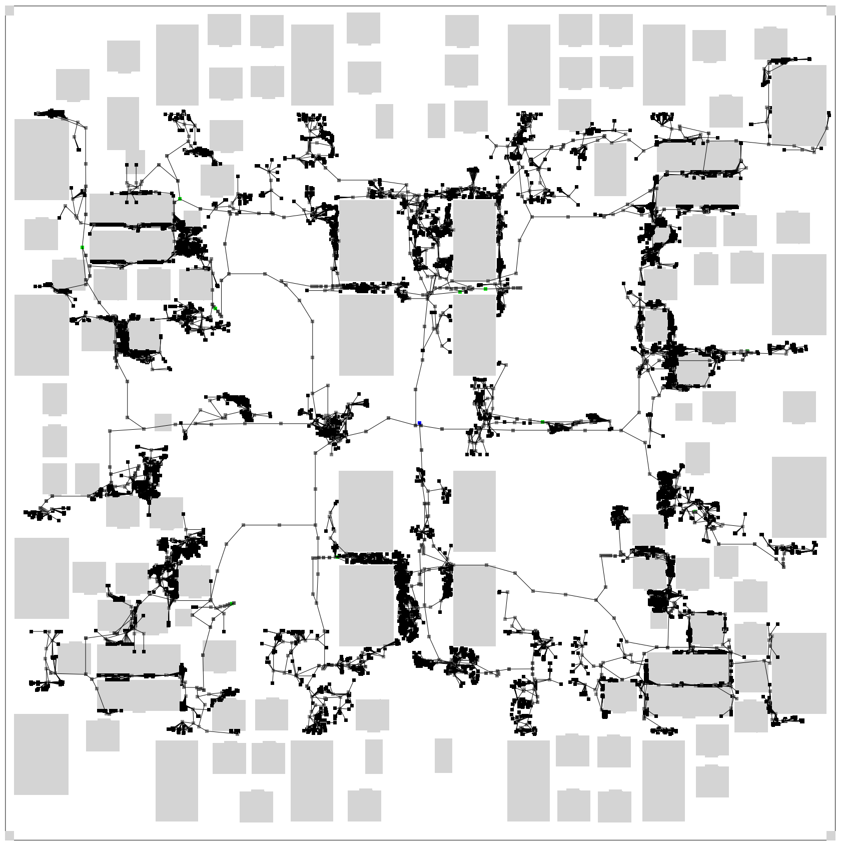

Figure 4: Clock A of 'Matthias' with 11 544 sinks. Each net is represented by a star.

[5] J. Cong, A.B. Kahng, C.-K. Koh, and C.-W.A. Tsao. Bounded-skew clock and steiner routing. *ACM Trans. on Design Automation of Electronic Systems 3*, 341–388, 1998.

[6] R. B. Deokar and S. S. Sapatnekar. A graph–theoretic approach to clock skew optimization. *Proc. of the IEEE Int. Symp. on Circuits and Systems*, 407–410, 1995.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1, 269–271, 1959.

[8] M. Edahiro. Minimum skew and minimum path length routing in vlsi layout design. NEC Research and Development, 32(4), 569-575, 1991.

[9] J. P. Fishburn. Clock skew optimization. *IEEE Trans. on Computers 39*, 945–951, 1990.

[10] S. Held. *Algorithmen für Potential-Balancierungs-Probleme und Anwendungen im VLSI-Design*. Diploma thesis. University of Bonn, 2001.

[11] R. B. Hitchcock, G. L. Smith, and D. D. Cheng. Timing analysis of computer hardware. *IBM Journal of Research and Development 26*, 100–105, 1982.

[12] I. S. Kourtev and E. G. Friedman. Simultaneous clock scheduling and buffered clock tree synthesis. *Proc. of the IEEE Int. Symp. on Circuits and Systems*, 1812–1815, 1997.

[13] I. S. Kourtev and E. G. Friedman. A quadratic programming approach to clock skew scheduling for reduced sensitivity to process parameter variations. *Proc. of the IEEE International ASIC/SOC Conference*, 210–215, 1999.

[14] I. S. Kourtev and E. G. Friedman. *Timing Optimization Through Clock Skew Scheduling*. Kluwer, Boston, 2000.

[15] M. Saitoh, M. Azuma, and A. Takahashi. Clustering based fast clock scheduling for light clock-tree. *Proc. DATE*, 240–244, 2001.

[16] K. A. Sakallah, T. N. Mudge, and O. A. Olukotun. checktc and mintc: Timing verification and optimal clocking of digital circuits. *Proc. ICCAD*, 552–555, 1990.

[17] N. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Graph algorithms for clock schedule optimization. *Proc. ICCAD*, 132–136, 1992.

[18] T. Szymanski. Computing optimal clock schedules. *Proc. DAC*, 399–404, 1992.

[19] C-W. A. Tsao and C-K. Koh. UST/DME: A clock tree router for general skew constraints. *Proc. ICCAD*, 400–405, 2000.

[20] R.-S. Tsay. An exact zero-skew clock routing algorithm. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 12*, 242–249, 1993.

[21] J. Vygen. *Theory of VLSI Layout*. Habilitation thesis. University of Bonn, 2001.

[22] J.G. Xi and W.-M. Dai. Useful-skew clock routing with gate sizing for low power design. *Journal of VLSI Signal Processing 16*, 163–179, 1997.

[23] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster parametric shortest path and minimum-balance algorithms. *Networks*, 21, 205–221, 1991.

chipname	Jens	Matthias	Katrin	Alex
technology	SA27E	SA27E	Cu11	Cu11
L_{drawn} (μm)	0.15	0.15	0.10	0.10
voltage (V)	1.8	1.8	1.2	1.5
image size (mm×mm)	2.29×2.00	13.80×13.88	13.80×13.88	9.31×9.38
# circuits	73 234	582 713	916 345	1 074 697
# pins	260 322	2 493 406	3 179 184	4 018 180
# nets	74 032	614 138	839 819	1 042 025
# clock domains	1	5	2	164
# sinks (registers)	3 805	128 292	137 218	194 208
biggest clock domain	3 805	55 826	137 135	48 667
# edges in slack balance graph	652 476	7 217 732	8 335 958	14 207 830
ltarget (ns)	1.00	0.40	1.50	0.30
etarget (ns)	0.00	0.05	0.00	0.05
ctarget (ns)	0.60	0.60	0.55	0.50
maximum allowed skew (ns)	0.60	0.60	0.80	0.20 - 1.20
$maxdist$ (mm)	0.3	0.3 - 0.6	0.7	0.25 - 1.8
$maxslew$ (ns)	0.400	0.400	0.630 / 0.400	0.100 - 0.250
# inserted clocksplitters	447	4 473	3 381 / 5 654	22 830
# inserted inverters	235	3 548	1 430 / 1 500	21 583
max. time an interval is missed (ns)	0.091	0.006	0.064 / 0.080	0.079
optimal worst slack (ns)[1]	0.721	0.290	1.233	-0.136
worst slack BC (ns)	0.790	0.272	0.977 / 1.145	-0.028 (-0.001)
worst slack ZS (ns)[1]	0.490	-0.175	0.310	-0.954 (-0.315)
worst slack ZST (ns)	-	-0.190	0.345	-
optimal frequency (MHz)[1]	159.3	269.5	147.8	802
maximum frequency BC (MHz)	161.0	268.2	142.4 / 145.9	900
maximum frequency ZST (MHz) [2]	153.6	238.7	130.6	701
difference BC / ZST [2]	+4.8%	+12.4%	+9.0% / +11.7%	+28.4%
worst slew BC (ns)	0.408	0.432	0.643 / 0.438	0.435
worst slew ZST (ns)	-	0.700	0.991	-
average slew BC (ns)	0.312	0.344	0.544 / 0.349	0.181
average slew ZST (ns)	-	0.362	0.565	-
area consumption by wiring BC (mm×mm)	0.244	1.423	1.928 / 2.376	7.989
area consumption by wiring ZST (mm×mm)	-	2.846	2.478	-
power consumption BC (W)	0.081	2.173	0.416 / 0.508	10.833
power consumption ZST (W)	-	2.906	0.441	-
difference BC / ZST	-	-25.2%	-5.7% / +15.3%	-
runtime late balancing (hh:mm:ss)	00:00:11	00:01:15	00:02:06	01:27:35
runtime early balancing	00:00:06	00:00:40	00:01:12	00:42:16
runtime intervals balancing	00:31:00	00:07:02	00:01:18	05:00:36
total runtime scheduling [3]	00:33:01	00:24:15	01:01:15	07:38:46
runtime clocktree construction	00:02:57	01:00:58	02:21:06 / 02:01:35	05:48:46
total runtime	00:35:58	01:25:13	03:22:21 / 03:02:50	13:27:32

Table 1: Computational results. BC = our clocktree construction, ZS = ideal zero-skew, ZST = zero-skew clocktree

[1]assuming 300ps on-chip variation in the clocktree
[2]for Jens and Alex the ZS results are used
[3]including pre- and post-optimization timing analysis