BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip

Bernhard Korte, Dieter Rautenbach, and Jens Vygen

***Abstract*— The BonnTools provide innvovative solutions for layout and timing closure that are used for many of the most complex integrated circuits. During 20 years of cooperation between the University of Bonn and IBM, new mathematical foundations and algorithms have been developed for the need of new technologies and leading-edge designs. In this paper we present the main ideas for placement, routing, timing optimization, and clock tree synthesis, which are the foundation of a continuing success story.**

***Index Terms*— physical design, layout, placement, routing, timing optimization, clock tree synthesis**

I. Introduction

The rapid development of VLSI technology, the abundance of interesting and clearly defined optimization problems arising in various design steps, the huge and exponentially increasing instance sizes, and the economic relevance make VLSI design a most appealing application area of mathematics.

The Research Institute for Discrete Mathematics at the University of Bonn has been working on problems arising in VLSI design for twenty years. Since 1987 there exists an intensive and growing cooperation with IBM, in the course of which more than one thousand chips of IBM and its customers have been designed with BonnTools. These contain complete solutions for placement, timing closure, clock tree synthesis, and routing, which have been developed in Bonn and are being used in many design centers all over the world. In 2005 the cooperation was extended to include Magma Design Automation. BonnTools are now also part of Magma's products and are used by its customers.

The distinguishing feature of BonnTools is their innovative mathematics. Almost all classical combinatorial optimization problems arise at some stage in VLSI design (cf. [26], [25]), and very efficient algorithms for these problems can be used to solve various subproblems in the design flow. However, many problems do not fit into standard patterns and need new customized algorithms. Many such algorithms have been developed by our group in Bonn and are now part of the design flow. By new technological challenges, new orders of magnitude in instance sizes, and new foci on objectives like power or yield, new problems arise constantly and classical problems require new solutions. This makes this field most interesting not only for engineers, but also for mathematicians.

In this paper we describe the key mathematical components of BonnTools. They are all used intensively for complex industrial chips. Many microprocessor series and hundreds of ASICs, including the most complex system-on-a-chip (SoC) designs, have been designed with these tools. In almost all cases the design is not done in a hierarchical mode, but with millions of movable objects on the top level, a few of which are large macros representing memory or logic cores or analog components. This almost flat design style allows for better solutions and decreases design cost and time-to-market, but poses challenges to running times of algorithms in order to meet tight turn-around-time requirements.

This paper is organized as follows. First, in Section II, we describe our placement tool BonnPlace and its key algorithmic ingredients. Global placement uses quadratic placement and a new multisection algorithm. Detailed placement is based on a sophisticated minimum cost flow formulation.

In Section III we proceed to timing optimization, where we concentrate on the three most important topics: repeater trees, logic restructuring, and choosing physical realizations of gates (sizing and V_t-assignment). These are the main components of BonnTimeOpt, and each uses very new mathematical theory.

As described in Section IV, BonnCycleOpt further optimizes the timing and robustness by enhanced clock skew scheduling. It computes a time interval for each clock input of a storage element. BonnClock, our tool for clock tree synthesis, constructs clock trees meeting these time constraints and minimizing power consumption.

Finally, Section V is devoted to routing. Our router, BonnRoute, contains the first global router that directly considers timing, power, and yield, and is provably close to optimal. The unique feature of our detailed router is an extremely fast implementation of Dijkstra's shortest path algorithm, allowing us to find millions of shortest paths even for long-distance nets in very reasonable time.

II. Placement

BonnPlace consists of global and detailed placement. Global placement ends with an infeasible placement, but with overlaps that can be removed by local moves: there is no large region that contains too many objects. Detailed placement, or legalization, takes the global placement as input and legalizes it by making only local changes.

Our global placement has two major components: quadratic placement and multisection.

At each stage the chip area $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ is partitioned by coordinates $x_{\min} = x_0 \leq x_1 \leq x_2 \leq \ldots \leq x_{n-1} \leq x_n = x_{\max}$ and $y_{\min} = y_0 \leq y_1 \leq y_2 \leq \ldots \leq y_{m-1} \leq y_m = y_{\max}$ into an array of regions $R_{ij} = [x_{i-1}, x_i] \times [y_{j-1}, y_j]$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Initially, $n = m = 1$. Each movable object is assigned to one region (cf. Figure 1).

The authors are with the Research Institute for Discrete Mathematics, University of Bonn, Lennéstr. 2, 53113 Bonn, Germany
Manuscript received April 12, 2006; revised August 31, 2006

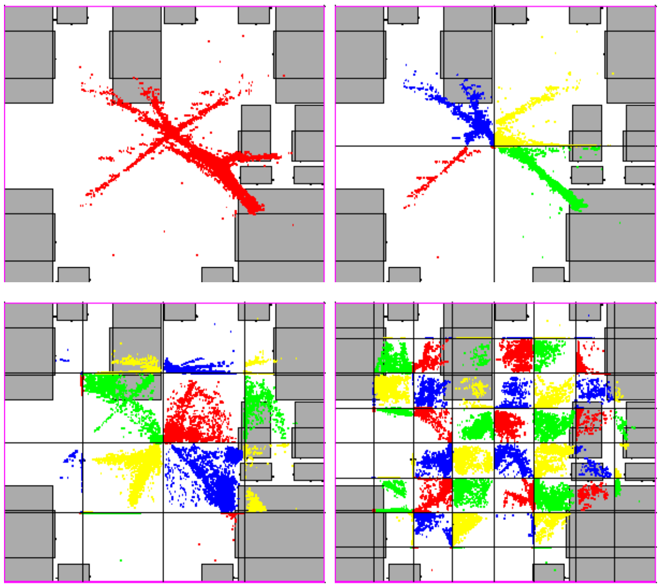

Fig. 1. The initial four levels of the global placement with 1, 4, 16, and 64 regions. Colors indicate the assignment of the movable objects to the regions.

In the course of global placement, columns and rows of this array, and thus the regions, are subdivided, and movable objects are assigned to subregions. After global placement, these rows correspond to circuit rows with the height of standard cells, and the columns are small enough so that no region contains more than a few dozen movable objects. On a typical chip in 65 nm technology we have, depending on the library and die size, about 5000 rows and 1000 columns.

A. *Quadratic Placement*

Quadratic placement means solving

$$\min \sum_{N\in\mathcal{N}} \frac{w(N)}{|N|-1} \sum_{p,q\in N} (X_{p,q}+Y_{p,q}),$$

where $\mathcal{N}$ is the set of nets, each net N is a set of pins, $|N|$ is its cardinality (which we assume to be at least two), and $w(N)$ is the weight of the net, which can be any positive number. For two pins p and q of the same net, $X_{p,q}$ is the function

(i) $(x(C)+x(p)-x(D)-x(q))^2$ if p belongs to movable object C with offset $x(p)$, q belongs to movable object D with offset $x(q)$, and C and D are assigned to regions in the same column.

(ii) $(x(C)+x(p)-v)^2$ if p belongs to movable object C with offset $x(p)$, C is assigned to region $R_{i,j}$, q is fixed at a position with x-coordinate u, and $v = \max\{x_{i-1}, \min\{x_i, u\}\}$.

(iii) $(x(C)+x(p)-x_i)^2 + (x(D)+x(q)-x_{i'-1})^2$ if p belongs to movable object C with offset $x(p)$, q belongs to movable object D with offset $x(q)$, C is assigned to region $R_{i,j}$, D is assigned to region $R_{i',j'}$, and $i < i'$.

(iv) 0 if both p and q are fixed.

$Y_{p,q}$ is defined analogously, but with respect to y-coordinates, and with rows playing the role of columns.

In its simplest form, with $n = m = 1$, quadratic placement gives coordinates that optimize the weighted sum of squares of Euclidean distances of pin-to-pin connections (cf. the top left part of Figure 1). Replacing multiterminal nets by cliques (i.e. considering a connection between p and q for all $p, q \in N$) is the best one can do, as was shown in [10]. Dividing the weight of a net by $|N| - 1$ is necessary to prevent large nets from dominating the objective function. Splitting nets along cut coordinates as in (ii) and (iii), first proposed in [42], partially linearizes the objective function and reflects the fact that long nets will be buffered later.

There are several reasons for optimizing this quadratic objective function. Firstly, delay along unbuffered wires grows quadratically with the length. Secondly, quadratic placement yields unique positions for most movable objects, allowing one to deduce much more information than the solution to a linear objective function would yield. Thirdly, as shown in [47], quadratic placement is stable, i.e. almost invariant to small netlist changes. Finally, quadratic placement can be solved extremely fast.

To compute a quadratic placement, first observe that the two independent quadratic forms, with respect to x- and y-coordinates, can be solved independently in parallel. Moreover, each row and column can be considered separately and in parallel. We solve each quadratic program by the conjugate gradient method with incomplete Cholesky preconditioning. The running time depends on the number of variables, i.e. the number of movable objects, and the number of nonzero entries in the matrix, i.e. the number of pairs of movable objects that are connected. As large nets result in a quadratic number of connections, we replace large cliques, i.e. connections among large sets of pins in the same net that belong to movable objects assigned to regions in the same column (or row when considering y-coordinates), equivalently by stars, introducing a new variable for the center of a star. This has been proposed in [42] and [8].

The running time to obtain sufficient accuracy grows slightly faster than linearly. There are linear-time multigrid solvers, but they do not seem to be faster in practice. We can compute a quadratic placement within at most a few minutes for 5 million movable objects. This is for the unpartitioned case $n = m = 1$; the problem becomes easier by partitioning, even when sequential running time is considered.

It is probably not possible to add linear inequality constraints to the quadratic program without a significant impact on the running time. However, linear equality constraints can be added easily, as was shown by [22]. Before partitioning, we analyze the quadratic program and add center-of-gravity constraints to those regions whose movable objects are not sufficiently spread. As the positions are the only information considered by partitioning, this is necessary to avoid random decisions.

B. *Multisection*

Quadratic placement usually has many overlaps which cannot be removed locally. Before legalization we have to ensure that no large region is overloaded. For this our global placement has a second main ingredient, which we call multisection.

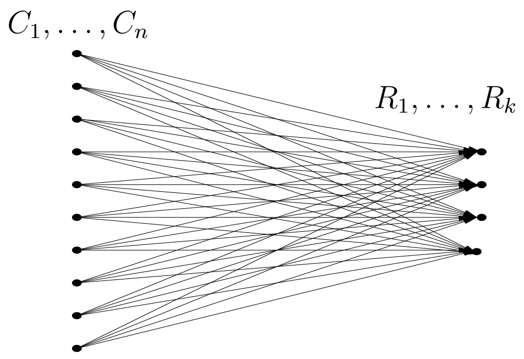

Fig. 2. Modeling multisection as a Hitchcock transportation problem. All arcs are oriented from left to right and are uncapacitated. Vertices on the left correspond to movable objects and have supply $a_1, \ldots, a_n$. Vertices on the right correspond to subregions and have demand $b_1, \ldots, b_k$. The cost of an arc (C_i, R_j) is $d(i, j)$. Note that $k \ll n$.

The basic idea is to partition a region and assign each movable object to a subregion. While capacity constraints have to be observed, the total movement should be minimized, i.e. the positions of the quadratic placement should be changed as little as possible.

More precisely, let $C_1, \ldots, C_n$ be the movable objects in a region, with sizes $a_1, \ldots, a_n$. Let $R_1, \ldots, R_k$ be the subregions with capacities $b_1, \ldots, b_k$, and let $d(i, j)$ denote the cost of moving C_i to R_j. Then we look for an assignment $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$ such that $\sum_{i:f(i)=j} a_i \leq b_j$ for $j = 1, \ldots, k$ and $\sum_{i=1}^{n} d(i, f(i))$ is minimum.

This partitioning strategy has been proposed in [42] for $k = 4$, and then generalized to arbitrary k in [8]. The problem is *NP*-hard, but it suffices to solve the fractional relaxation, where we look for $g : \{1, \ldots, n\} \times \{1, \ldots, k\} \to [0, 1]$ such that $\sum_{j=1}^{k} g(i, j) = 1$ for $i = 1, \ldots, n$, $\sum_{i=1}^{n} g(i, j)a_i \leq b_j$ for $j = 1, \ldots, k$, and $\sum_{i=1}^{n} \sum_{j=1}^{k} g(i, j)d(i, j)$ is minimum. The reason is that from any optimum fractional solution an almost integral one, with at most $k - 1$ fractionally assigned movable objects, can easily be obtained [45].

This fractional relaxation is a Hitchcock transportation problem (cf. Figure 2), and can thus be solved by standard minimum cost flow algorithms (cf. [26]). However, these have a superquadratic running time and are too slow. For the quadrisection case, where $k = 4$ and d is the ℓ_1-distance, we described a linear-time algorithm in [45], which is quite complicated but very efficient. For the general case Brenner [5] recently proposed an $O(nk^2(\log n + k \log k))$-algorithm. This is extremely fast also in practice and has replaced the quadrisection algorithm of [45] in BonnPlace.

The idea is based on the well-known successive shortest paths algorithm (cf. [26]). Assume $a_1 \geq a_2 \geq \cdots \geq a_n$. We assign the objects in this order. A key observation is that for doing this optimally we need to re-assign only $O(k^2)$ previously assigned objects and thus can apply a minimum cost flow algorithm in a digraph whose size depends on k only. Note that k is less than 10 in all our applications, while n can be in the millions.

Figure 3 shows a multisection example where the movable objects are assigned optimally to nine regions.

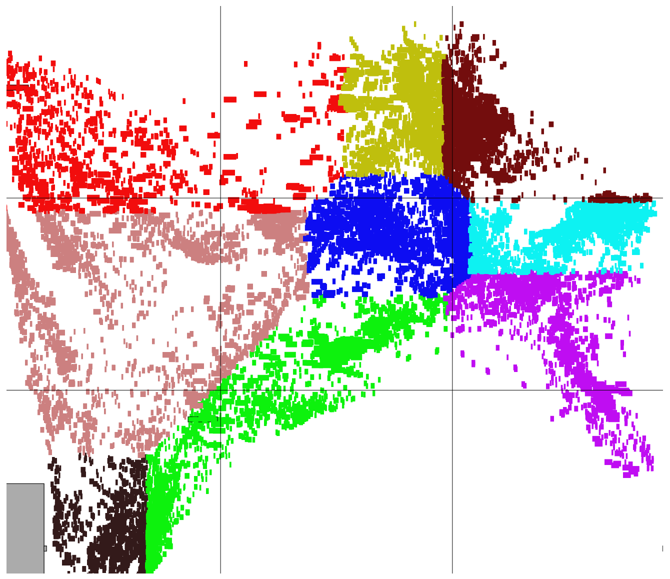

Fig. 3. Example for multisection: objects are assigned to 3×3 subregions. The colors reflect the assignment: the red objects are assigned to the top left region, the yellow ones to the top middle region, and so on. This assignment is optimal with respect to total ℓ_1-distance.

C. Overall Global Placement

With these two components, quadratic placement and multisection, our global placement can be described. Each level begins with a quadratic placement. Before subdividing the array of regions further, we fix macro cells that are too large to be assigned completely to a subregion. Our macro placement uses minimum cost flow, branch-and-bound, and greedy techniques. Interaction of small and large blocks in placement is still not fully understood, and placing large macros in practice typically requires a significant amount of manual interaction.

After partitioning the array of regions, the movable objects are assigned to the resulting subregions. Several strategies are applied (see [8] for details), but the core subroutine in each case is the multisection described above. An important further step is repartitioning, where 2×2 or even 3×3 subarrays of regions are considered and all their movable objects are reassigned to these regions, essentially by computing a local quadratic placement followed by multisection.

There are further components which reduce routing congestion [7], deal with timing and resistance constraints, and handle other constraints like user-defined bounds on coordinates or distances of some objects. Global placement ends when the rows correspond to cell rows. Typically there are fewer columns than rows as most movable objects are wider than high. Therefore we often use 2×3 partitioning in the late stages of global placement.

D. Detailed Placement

Detailed placement, or legalization, considers standard cells (movable objects of unit height) only; all others are fixed beforehand. The task is to place the standard cells legally without changing the (illegal) input placement too much. It

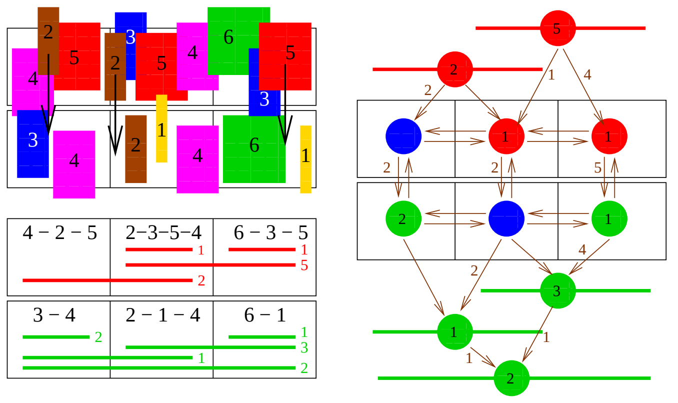
Fig. 4. An example with two zones and six regions, each of width 10 (top left), the supply (red) and demand (green) regions and intervals with their supply and demand (bottom left), and the minimum cost flow instance (right) with a solution shown in brown numbers. To realize this flow, objects of size 2, 2, and 5, respectively, have to be moved from the top regions downwards.

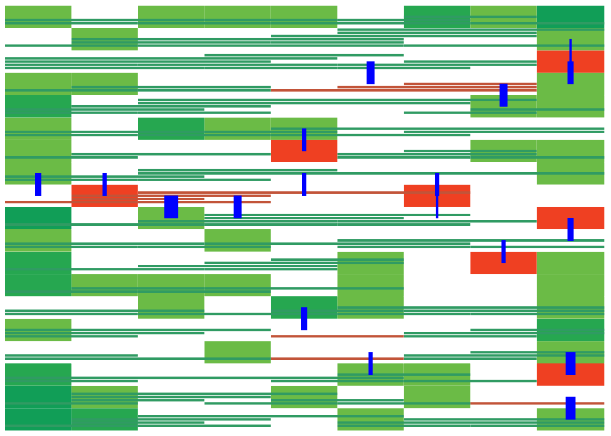
Fig. 5. Small part of a real chip in legalization. Supply regions and intervals are shown in red, demand regions and intervals in green. The blue edges represent the minimum cost flow, and their width is proportional to the amount of flow.

is quite natural to model this problem as a minimum cost flow problem, where flow goes from supply regions with too many objects to demand regions with extra space [43]. We have refined this approach in [11] and describe this enhanced legalization algorithm, which is part of BonnPlace, in the following.

It consists of three phases. By a zone we mean a maximal part of a cell row that is not blocked by any fixed objects, i.e. can be used for legalization. The first phase guarantees that no zone contains more cells than fit into it. The second phase places the cells legally within each zone in the given order. When minimizing quadratic movement, this can be done optimally in linear time, as shown in [11] (see also [20] and [9]). Finally, some post-optimization heuristics (like exchanging two cells, but also much more complicated operations) are applied.

The most difficult and important phase is the first one. If the global placement is very dense in some areas, a significant number of cells have to be moved. As phase two works in each zone separately, phase one has to guarantee that no zone contains more objects than fit into it.

In order to prevent large distance movements within the zones in phase two, wide zones are partitioned into regions. Each movable object is assigned to a region. Unless all movable objects that are assigned to a region R can be placed legally with their center in R, some of them have to be moved out of R. But this is not sufficient: in addition, it may be necessary to move some objects out of certain sequences of consecutive regions. More precisely, for a sequence of consecutive regions $R_1, \dots, R_k$ within a zone, we define its supply by

$$\begin{aligned} supp(R_1,\dots,R_k) &:= \\ &\max\Bigg\{0,\ \sum_{i=1}^{k}(w(R_i)-a(R_i)) - \frac{1}{2}(w_l(R_1)+w_r(R_k)) \\ &\qquad - \sum_{1\le i<j\le k,(i,j)\neq(1,k)} supp(R_i,\dots,R_j)\Bigg\}, \end{aligned}$$

where $a(R_i)$ is the width of region R_i, $w(R_i)$ is the total width of cells that are currently assigned to region R_i, and $w_l(R_i)$ and $w_r(R_i)$ are the widths of the leftmost and rightmost cell in R_i, respectively, or zero if R_i is the leftmost (rightmost) region within the zone.

If $supp(R_1,\dots,R_k)$ is positive, $(R_1,\dots,R_k)$ is called a supply interval. Similarly, we define the demand of each sequence of consecutive regions, and the demand intervals. The regions, supply intervals and demand intervals form a digraph in which we compute a minimum cost flow that cancels demands and supplies. The construction of this minimum cost flow instance is illustrated in Figure 4. Figure 5 shows a typical result on a real chip.

Finally the flow is realized by moving objects along flow arcs. We scan the arcs carrying flow in topological order and solve a multi-knapsack problem by dynamic programming for selecting the best set of cells to be moved for realizing the flow on each arc.

The minimum cost flow formulation yields an optimum solution under some assumptions, and an excellent one in practice. Experimental results show that the gap between a computed solution and a theoretical lower bound is only approximately 10%, and neither timing nor routability is significantly affected [6].

III. Timing Optimization

In this section we describe the main ingredients of BonnTimeOpt, our timing optimization routines. These include algorithms for the construction of timing- and routing-aware fanout trees (repeater trees), for the timing-oriented logic restructuring and optimization, and for the timing- and power-aware choice of different physical realizations of individual gates. Each is based on new mathematical theory.

Altogether, these routines combined with appropriate net weight generation and iterative placement runs form the so-called fast timing-driven placement loop, our solution for timing closure. Using these new very fast subroutines, we managed to decrease the overall turn-around time for timing closure, including full placement and timing optimization, from more than a week to 26 hours on the largest designs.

A. Fanout Trees

On an abstract level the task of a fanout tree is to carry a signal from one gate, the root r of the fanout tree, to other gates, the sinks $s_1, \dots, s_n$ of the fanout tree, as specified by the netlist. If the involved gates are not too numerous and not too far apart, then this task can be fulfilled just by a metal connection of the involved pins, i.e. by a single net without any repeaters. But in general we need to insert repeaters (buffers or inverters).

In fact, fanout trees are a very good example for the observation mentioned in the introduction that the development of technology continually creates new complex design challenges that also require new mathematics for their solution. Whereas circuit delay traditionally dominated the interconnect delay and the construction of fanout trees was of secondary importance for timing, the feature size shrinking is about to change this picture drastically. Extending the current trends one can predict that in future technologies more than half of all circuits of a design will be needed just for bridging distances, i.e. in fanout trees.

An instance of the repeater tree problem consists of (i) the arrival time $AT(r)$ at the root r and a required arrival time $RAT(s)$ at each sink s, (ii) a parity in $\{+,-\}$ for each sink indicating whether it requires the signal or its inversion, (iii) placement information for the root and the sinks $Pl(r), Pl(s_1), Pl(s_2), ..., Pl(s_n) \in [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, (iv) physical information about the driver strength of r and the input capacitances $InputCap(s_i)$ of the sinks, and (v) physical information about the wiring and the library of available repeaters.

The procedure that we propose for fanout tree construction [4] works in two phases. The criticality of the individual sinks is estimated by taking their required signal arrival times, their input capacitances, their distance from the root, and the driver strength of the root into account. The first phase generates a preliminary topology for the fanout tree, which connects very critical sinks in such a way as to maximize the minimum slack, and which minimizes wiring for non-critical sinks. During the second phase the resulting topology is finalized and buffered in a bottom-up fashion using mainly inverters and respecting the parities of the sinks.

In order to quantify the criticality of an individual sink s, we estimate the slack σ_s that arises at s if we connect s to r via an optimally buffered 2-terminal fanout tree. Since optimally buffering a 2-point connection approximately linearizes the delay as a function of the distance, we can consider the delay from r to s to be proportional to their distance and obtain

$$\begin{aligned}\sigma_s \quad := \quad & RAT(s) - AT(r) - c_{wire} dist(Pl(r), Pl(s)) \\ & -f_1(InputCap(s)) - f_2(r)\end{aligned}$$

where f_1 and f_2 are estimates of the delay effects of the input capacitance of s and the driver strength of r. We determine the involved constants and functions in a preprocessing step. The striking accuracy of this very simple delay model is illustrated in Figure 6, which compares the estimated delay with the measured delay after buffering and sizing.

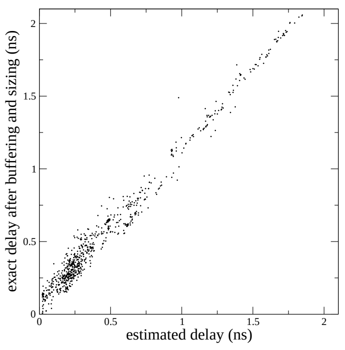

Fig. 6. The simple timing model used for topology generation matches actual timing results after buffering well.

The individual sinks are now inserted one by one into the preliminary topology in order of non-increasing criticality, i.e. non-decreasing value of σ_s. A preliminary topology is a pair (T, Pl) where T is an arborescence and $Pl : V(T) \to [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$ is an embedding of the vertices of T in the chip area. T is rooted at r and the leaves of T are precisely the sinks s_i. In T the root r has one child and all internal nodes have exactly two children.

When we insert a new sink s, we consider all arcs $e = (u, v) \in E(T)$ of the preliminary topology constructed so far and estimate the effect of subdividing e by a new internal node w and connecting s to w in a shortest possible way.

The additional wiring amounts to $l_e = dist(Pl(s), Pl(w))$.

In order to quantify the delay effects, one has to observe that the final fanout tree will contain some first gates on the paths from w to the sinks. These represent an additional capacitance. We model this by adding a delay contribution c_{node} to the estimated delays on the two branches emanating at w. c_{node} is determined during preprocessing and is about 10 to 20ps.

The sink s will be inserted in an arc e of T that maximizes $\xi\sigma_e - (100 - \xi)l_e$, where σ_e estimates the corresponding worst slack. The parameter $\xi \in [0, 100]$ allows us to favor slack maximization for timing critical instances or wiring minimization for non-critical instances. Figure 7 gives an example for a preliminary topology.

In most cases it is reasonable to choose values for ξ that are neither too small nor too large. Nevertheless, in order to mathematically validate our procedure we have proved optimality statements for the extreme values $\xi = 0$ and $\xi = 100$. If we ignore timing ($\xi = 0$), the final length of the topology is at most $3/2$ times the minimum length of a rectilinear Steiner tree connecting the root and the sinks. If we ignore wiring ($\xi = 100$), the topology realizes the optimum slack within our delay model (up to c_{node} for non-integral input) [4].

After inserting all sinks into the preliminary topology, the second phase begins, in which we insert the actual inverters. For each sink s we create a cluster C containing only s. In general a cluster C is assigned a position $Pl(C)$, a set of sinks $S(C)$ all of the same parity, and an estimate $W(C)$ for the

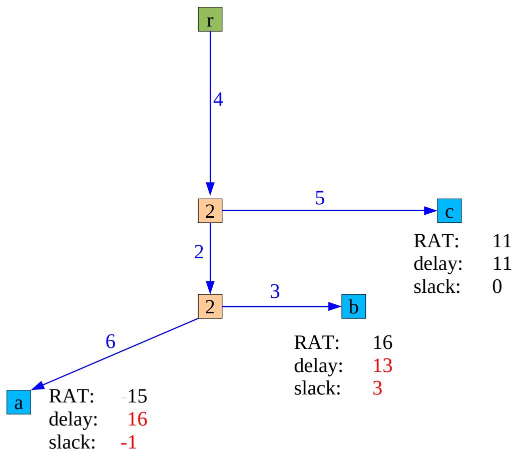

Fig. 7. An example for topology generation with $AT(r) = 0$, $c_{wire} = 1$, $c_{node} = 2$, $f_1 = f_2 = 0$, and three sinks a, b and c with displayed required arrival times. The criticalities are $\sigma_a = 15 - 0 - (4 + 2 + 6) = 3$, $\sigma_b = 16 - 0 - (4 + 2 + 3) = 7$, and $\sigma_c = 11 - 0 - (4 + 5) = 2$. Our algorithm first connects the most critical sink c to r. The next critical sink is a which is inserted into the only arc (r, c) creating an internal node w. For the insertion of the last sink b there are now three possible arcs (r, w), (w, a), and (w, c). Inserting b into (w, a) creates the displayed topology whose worst slack is -1, which is best possible here.

wiring capacitance of a net connecting a circuit at position $Pl(C)$ with the sinks in $S(C)$. The elements of $S(C)$ are either original sinks of the fanout tree or inverters that have already been inserted.

There are three basic operations on clusters. Firstly, if $W(C)$ and the total input capacitance of the elements of $S(C)$ reach certain thresholds, we insert an inverter I at position $Pl(C)$ and connect it by wire to all elements of $S(C)$. We create a new cluster C' at position $Pl(C)$ with $S(C') = \{I\}$ and $W(C) = 0$. As long as the capacitance thresholds are not attained, we can move the cluster along arcs of the preliminary topology towards the root r. By this operation $W(C)$ increases while $S(C)$ remains unchanged. Finally, if two clusters happen to lie on a common position and their sinks are of the same parity, we can merge them, but we may also decide to add inverters for some of the involved sinks. This decision again depends on the capacitance thresholds and on the objectives timing and wirelength.

During buffering, the root connects to the clusters via the preliminary topology and the clusters connect to the original sinks s_i via appropriately buffered nets. Once all clusters have been merged to one which arrives at the root r, the construction of the fanout tree is completed.

The optimality statements which we proved within our delay model and the final experimental results show that the second phase nearly optimally buffers the desired connections. Our procedure is extremely fast. The topology generation solved 4.6 million instances with up to 10000 sinks from a current 90 nm design in less than 100 seconds on a 2.6 GHz Opteron machine [4], and the buffering is completed in less than 10 minutes. On average we deviated less than 1.5 % from the minimum length of a rectilinear Steiner tree when minimizing wire length, and less than 2 ps from the theoretical upper slack bound when maximizing worst slack.

We are currently including enhanced buffering with respect to timing constraints, wire sizing, and plane assignment in our algorithm. We are also considering an improved topology generation, in particular when placement or routing resources are limited.

B. *Fanin Trees*

Whereas in the last section one signal had to be propagated to many destinations via a logically trivial structure, we now look at algorithmic tasks posed by the opposite situation in which several signals need to be combined to one signal as specified by some Boolean expression. The netlist itself implicitly defines such a Boolean expression for all relevant signals on a design. The decisions about these representations were taken at a very early stage in the design process, i.e. in logic synthesis, in which physical effects could only be crudely estimated. At a relatively late stage of the physical layout process much more accurate estimates are available. If most aspects of the layout have already been optimized but we still see negative slack at some gates, changing the logic that feeds the gate producing the late signal is among the last possibilities for eliminating the timing problem. Traditionally, late changes in the logic are a delicate matter and only very local modifications replacing some few gates have been considered, also due to the lack of global algorithms.

To overcome the limitations of purely local and conservative changes, we have developed a totally novel approach that allows for the redesign of the logic on an entire critical path taking all timing and placement information into account [35]. Whereas most procedures for Boolean optimization of combinational logic are either purely heuristic or rely on exhaustive enumeration and are thus very time consuming, our approach is much more effective.

Assume that we are given a critical path P which combines a number of signals $x_1, x_2, \ldots, x_n$ arising at certain times $AT(x_i)$ and locations $Pl(x_i)$ by a sequence $g_1, g_2, \ldots, g_m$ of gates such that g_j takes as inputs the output of g_{j-1} and some of the x_i and the output signal of g_m is required at a certain location within a given required arrival time.

Our algorithm first generates a standard format. It decomposes complex gates on P into elementary `and`- and `or`-gates with fanin two plus inversions. Applying the de Morgan rules we eliminate all inversions but those on input signals of P. We arrive at a situation in which P is essentially represented by a sequence of `and`- and `or`-gates. Equivalently, we could do with `nand`-gates only, and we will indeed use `nand`s for the final realization. However, for the sake of a simpler description of our algorithm, `and`- and `or`-gates are more suitable.

We now design an alternative, logically equivalent representation of the signal produced by g_m as a function of the x_i in such a way that late input signals do not pass through too many logic stages of this alternative representation. This is easy if this sequence consists either just of `and`-gates or just of `or`-gates. The most difficult case occurs if the `and`- and `or`-gates alternate, i.e. the function calculated by P is of the

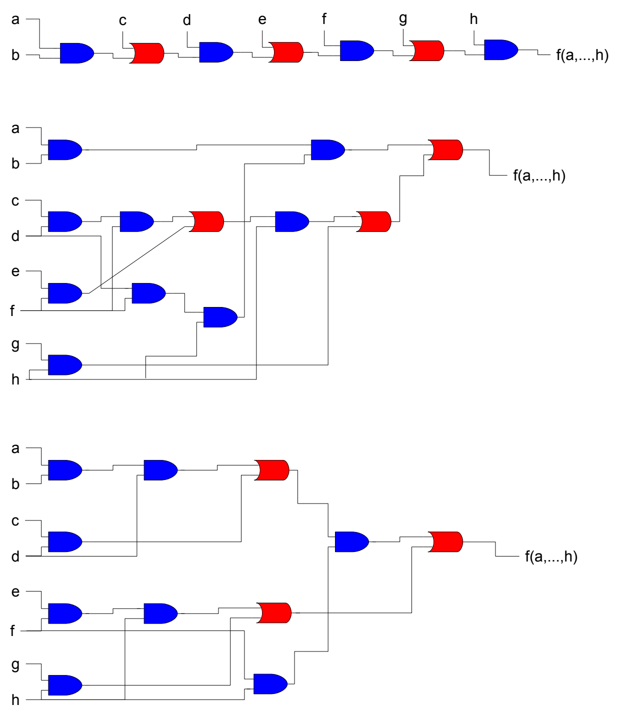

Fig. 8. Three logically equivalent circuits for the function $f(a,b,...,h)$ that correspond to the formulas $f(a,...,h) = ((((((a \wedge b) \vee c) \wedge d) \vee e) \wedge f) \vee g) \wedge h$, $f(a,...,h) = ((a \wedge b) \wedge ((d \wedge f) \wedge h)) \vee (((((c \wedge d) \wedge f) \vee (e \wedge f)) \wedge h) \vee (g \wedge h))$, and $f(a,...,h) = ((((a \wedge b) \wedge d) \vee (c \wedge d)) \wedge (f \wedge h)) \vee (((e \wedge f) \wedge h) \vee (g \wedge h))$. The first path is a typical input of our procedure and the two alternative netlists have been obtained by the dynamic programming procedure based on the identity (1). Ignoring wiring and assuming unit delays for the gates, the second netlist would for instance be optimal for $AT(a) = AT(b) = AT(g) = AT(h) = 3$, $AT(e) = AT(f) = 1$, and $AT(c) = AT(d) = 0$ leading to an arrival time of 6 for $f(a,...,h)$ instead of 10 in the input path.

form

$$\begin{aligned} & f(x_1, x_1', x_2, x_2', \dots, x_n, x_n') \\ & \quad := \quad ((\cdots(((x_1 \wedge x_1') \vee x_2) \wedge x_2') \cdots) \vee x_n) \wedge x_n' \\ & \quad = \quad \bigvee_{i=1}^{n} \left(x_i \wedge \left(\bigwedge_{j=i}^{n} x_j' \right) \right). \end{aligned}$$

In this case we apply dynamic programming based on identities like the following:

$$\begin{aligned} & f(x_1, \dots, x_n') \; = \\ & \left(f(x_1, \dots, x_l') \wedge \left(\bigwedge_{j=l+1}^{n} x_j' \right) \right) \vee \; f(x_{l+1}, \dots, x_n') \end{aligned} \tag{1}$$

Our dynamic programming procedure maintains sets of useful subfunctions such as $f(x_i, \dots, x_j')$ and $\bigwedge_{k=i}^{j} x_k'$ together with estimated timing and placement information. In order to produce the desired final signal, these sets of subfunctions are combined using small sets of gates, and the timing and placement information is updated. We maintain only those representations that are promising. The final result of our algorithm is found by backtracking through the data accumulated by the dynamic programming. After having produced a faster logical representation, we apply de Morgan rules once more and collapse several consecutive elementary gates to more complex ones if this improves the timing behaviour. In many cases this results in structures mainly consisting of `nand`-gates and inverters.

Our procedure is very flexible and contains the purely local changes as a special case. Whereas the dynamic programming procedure is quite practical and easily allows us to incorporate physical insight as well as technical constraints, we can validate its quality theoretically by proving interesting optimality statements.

For example, let $\epsilon > 0$ be arbitrarily small. If we neglect placement information, assume non-negative integer arrival times and further assume a unit delay for `and`- and `or`-gates, then the arrival time of the signal as calculated by our alternative realization is within a factor of $(1+\epsilon)$ of the best possible arrival time over all circuits using arbitrary gates with fanin two [37].

Besides the described procedure for logic optimization on critical paths we have developed theoretical machinery for designing complex subfunctions taking timing information into account [36].

C. Gate Sizing and V_t-Assignment

The two problems considered in this section consist of making individual choices from some discrete sets of possible physical realizations for each gate of the netlist such that some global objective function is optimized.

For gate sizing one has to determine the size of the individual gate measured for instance by its area or power consumption. This size affects the input capacitance and driver strength of the gate and therefore has an impact on timing. A larger gate typically decreases downstream delay and increases upstream delay.

Whereas the theoretically most well-founded approaches for the gate sizing problem rely on convex programming formulations [13], these approaches typically suffer from their algorithmic complexity and restricted timing model. In many, especially local situations, approaches that choose gate sizes heuristically can produce competitive results because it is much easier to incorporate local physical insight into heuristic selection rules than into a sophisticated convex program. In BonnTimeOpt we use both, a global formulation and convex programming for the general problem as well as heuristics for special purposes.

For the simplest form of the global formulation we consider a directed graph G which encodes the netlist of the design. For a set V_0 of nodes v we are given signal arrival times a_v and we must choose gate sizes $x = (x_v)_{v \in V(G)} \in [l,u] \subseteq \mathbb{R}^{V(G)}$ and arrival times for nodes not in V_0 minimizing $\sum_{v \in V(G)} x_v$ subject to the timing constraints $a_v + d_{(v,w)}(x) \leq a_w$ for all arcs $(v,w) \in E(G)$. The delay $d_{(v,w)}(x)$ of some arc (v,w) is modeled by an arbitrary linear function with positive coefficients depending on quotients of the form $\frac{x_w}{x_v}$. Dualizing the timing constraints via Lagrange multipliers $\lambda_{uv} \geq 0$, the dual optimality conditions imply that $(\lambda_e)_{e \in E}$ of an optimal solution constitutes a non-negative flow on the graph G [13].

For given dual variables the problem reduces to minimizing a weighted sum of the gate sizes x and delays $d_{uv}(x)$ subject to $x \in [l, u]$, which can be done by a simple iterative procedure with linear convergence rate [38]. The overall algorithm is the classical constrained subgradient projection method (cf. [28]). The known convergence guarantees for this algorithm require an exact projection, which means that we have to determine the above-mentioned non-negative flow on G that is closest to some given vector $(\lambda_e)_{e \in E}$.

Since this exact projection is actually the most time-consuming part, practical implementations use crude heuristics having unclear impact on convergence and quality. To overcome this limitation, we proved in [39] that the convergence of the algorithm is not affected by executing the projection in an approximate and much faster way. This results in a stable, fast, and theoretically well-founded implementation of the subgradient projection procedure for gate sizing.

The second optimization problem that we consider in this section is V_t-assignment. A physical consequence of feature size shrinking is that leakage power consumption represents a growing part of the overall power consumption of a chip. Increasing the threshold voltage of a circuit reduces its leakage but increases its delay. Modern libraries offer circuits with different threshold voltages. The optimization problem that we face is to choose the right threshold voltages for all circuits, which minimize the overall (leakage) power consumption while respecting timing restrictions.

We first consider a netlist in which every circuit is realized in its slowest and least-leaky version. We define an appropriate graph G whose arcs are assigned delays, and some of whose arcs correspond to circuits for which we could choose a faster yet more leaky realization. For each such arc e we can estimate the power cost c_e per unit delay reduction. We add a source node s joined to all primary inputs and to all output nodes of memory elements and a sink node t joined to all primary outputs and to all input nodes of memory elements. Then we perform a static timing analysis on this graph and determine the set of arcs E' that lie on critical paths.

The general step now consists in finding a cheapest s-t-cut $(S, \bar{S})$ in $G' = (V(G), E')$ by a max-flow calculation in an auxiliary network. Arcs leaving S that can be made faster contribute c_e to the cost of the cut, and arcs entering S that can be made slower contribute $-c_e$ to the cost of the cut. Furthermore, arcs leaving S that cannot be made faster contribute ∞ to the cost of the cut, and arcs entering S that cannot be made slower contribute 0 to the cost of the cut.

If we have found such a cut of finite cost, we can improve the timing at the lowest possible power cost per time unit by speeding up the arcs from S to $\bar{S}$ and slowing down (if possible) the arcs from $\bar{S}$ to S. This optimality statement is proved in [32] subject to the simplifying assumptions that the delay/power dependence is linear and that we can realize arbitrary V_t-values within a given interval, which today's libraries typically do not allow. Nevertheless, the linearity of the delay/power dependence approximately holds locally and the discrete choosable values are close enough.

We point out that the described approach is not limited to V_t-assignment. It can be applied whenever we consider roughly independent and local changes and want to find an optimal set of operations that corrects timing violations at minimum cost. This has been part of BonnTools for some time [14], but previously without using the possibility of slowing arcs from $\bar{S}$ to S, and thus without optimality properties.

IV. Clock Scheduling and Clocktree Construction

Most computations on chips are synchronized. Each storage element (register, flip-flop, latch) receives a periodic clock signal, controlling the times when the bit at the data input is to be stored and transferred to further computations in the next cycle. Today it is well-known that striving for simultaneous clock signals (zero skew), as most chip designers did for a long time, is not optimal. By clock skew scheduling, i.e. by choosing individual clock signal arrival times for the storage elements, one can improve the performance. However, this also makes clock tree synthesis more complicated. For nonzero skew designs it is very useful if clock tree synthesis does not have to meet specified points in time, but rather time intervals. We proposed this methodology together with new algorithms in [2], [3], and [18]. Here we describe the basic ideas underlying BonnCycleOpt and BonnClock, the tools realizing this solution.

A. Clock Skew Scheduling

Let us define the latch graph as the digraph whose vertex set is the set of all storage elements and which contains an arc (x, y) if the netlist contains a path from the output of x to the input of y. Let $d(x, y)$ denote the maximum delay from x to y. If all storage elements have the same frequency $\frac{1}{T}$ (i.e., their cycle time is T), then a zero skew solution is feasible only if all delays are at most T. With clock skew scheduling one can relax this condition. In other words, for given delays one can improve the performance. In this simple case, we ask for arrival times $a(x)$ of clock signals at all storage elements x such that $a(x) + d(x, y) \leq a(y) + T$ holds for each arc (x, y) of the latch graph. Such arrival times exist if and only if the latch graph has no directed cycle such that the mean delay of its arcs is greater than T [3]. The optimal feasible cycle time T and feasible clock signal arrival times $a(t)$ can be computed by minimum mean cycle algorithms, e.g. those of Karp [21] and Young, Tarjan, and Orlin [48].

This simple situation is unrealistic. Today's systems on a chip have multiple frequencies and often several hundred different clock domains. The situation is further complicated by transparent latches, user-defined timing tests, and various advanced design methodologies.

Moreover, it is not sufficient to maximize the frequency only. The delays that are input to clock skew scheduling are necessarily estimates: detailed routing will be done later and will lead to different delays. Thus one would like to have as large a safety margin (roughly equivalent to positive slack; cf. [46]) as possible. In other words, the available slack should be distributed carefully, and the slack histogram (cf. Figure 9) should be lexicographically optimal.

Next, signals can also be too fast, and although such early-mode violations can be repaired by buffering, this can be very

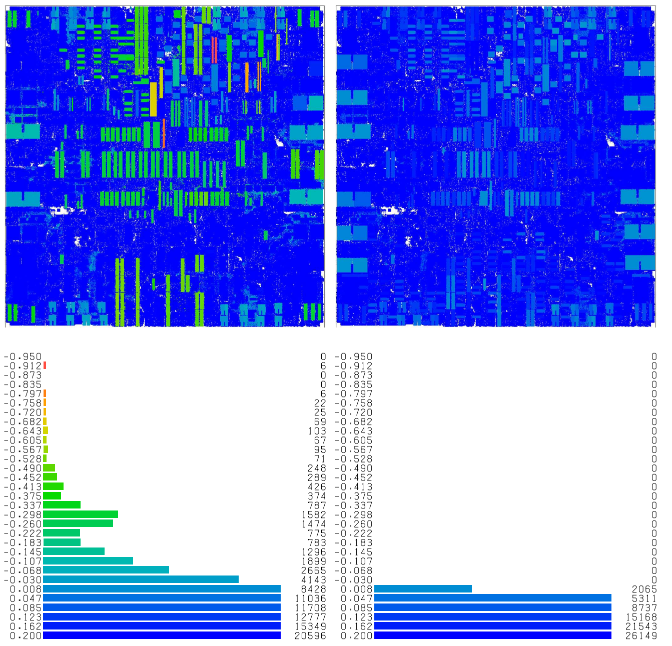

Fig. 9. Slack histograms showing the improvement due to clock skew scheduling and appropriate clock tree synthesis; left: zero skew, right: with BonnClock trees. Each histogram row represents a slack interval (in ns) and shows the number of slacks in this range. The placements on top are also colored according to these slacks.

expensive, and clock skew scheduling can remove most early-mode violations at almost no cost.

Finally, it is very hard to realize arbitrary individual arrival times exactly; moreover this would lead to high power consumption in clock trees. Computing time intervals rather than points in time is much better. Without making critical paths any worse, the power consumption (and use of space and wiring resources) by clock trees can be reduced drastically.

We have therefore proposed a three-stage clock skew scheduling approach in [3]. First, only late-mode slacks are considered. More precisely, we consider only those slacks that cannot be increased by inserting extra delays (user-defined timing tests may imply that this set is different from the set of late-mode slacks). Then we reduce early-mode violations (more precisely, slacks that can be increased by inserting extra delays), without decreasing any small or negative late-mode slacks. Thirdly, we compute a time interval for each storage element such that whenever each clock signal arrives within the specified time interval, no small or negative slack will decrease.

In the next section we discuss how to balance a certain set of slacks while not decreasing others.

B. Slack Balancing Models and Algorithms

In [3] and [17], generalizing the early work of Schneider and Schneider [40] and Young, Tarjan and Orlin [48], we have developed slack balancing algorithms for very general situations. The most general problem can be formulated as follows. Given a directed graph G (the timing graph), $d : E(G) \to \mathbb{R}$ (delays), a set $F_0 \subseteq E(G)$ (arcs where we are not interested in positive slack) and a partition $\mathcal{F}$ of $E(G) \setminus F_0$ (groups of arcs in which we are interested in the worst slack only), and weights $w : E(G) \setminus F_0 \to \mathbb{R}_{>0}$ (sensitivity of slacks), the task is to find arrival times $\pi : V(G) \to \mathbb{R}$ with $\pi(x) + d(e) \leq \pi(y)$ for $e = (x, y) \in F_0$ such that the vector of relevant slacks

$$\left(\min\left\{\frac{\pi(y) - \pi(x) - d(e)}{w(e)} \,\middle|\, e = (x,y) \in F\right\}\right)_{F \in \mathcal{F}}$$

(after sorting entries in non-decreasing order) is lexicographically maximal. In [46] we justified this model theoretically. Note that the delays d include cycle adjusts and thus can be negative (for an internal arc $e = (x, y)$ of a normal flip-flop, $d(e)$ is the propagation delay minus the cycle time). The conditions for $e \in F_0$ correspond to standard timing propagation rules.

The problem can be solved in $O(\min\{n^4 \log^2 n + n^2 m \log m,\ n^4 \log n + n^2 m \log^2 n \log\log n,\ w_{\max}(mn + n^2 \log n)\})$ time in general [17] and in $O(mn + n^2 \log n)$ time for unit weights [3]. In practice it can be solved much faster if we replace $\frac{\pi(y)-\pi(x)-d(e)}{w(e)}$ by $\min\{\Theta, \pi(y) - \pi(x) - d(e)\}$, i.e. ignore slacks beyond a certain threshold Θ, which we typically set to 50ps for early-mode slacks and 300ps for late-mode slacks.

Positive slacks which have been obtained previously and should not be decreased can be modeled simply by increasing the corresponding delays. Time intervals for clock signal arrival times also correspond to positive slack on the arcs corresponding to storage elements.

The basic algorithm iteratively determines the most critical cycle and contracts it. By working on the timing graph rather than on the latch graph, we can consider all complicated timing constraints, different frequencies, etc. directly. On the other hand, contracting parts of the timing graph efficiently is not easy. In our experiments it turned out to be most efficient to use a combination of the latch graph and the timing graph, incorporating the advantages of both models.

Figure 9 shows a typical result on a leading-edge ASIC. The left-hand side shows the slacks after timing-driven placement, but without clock skew scheduling, assuming zero skew and estimating the on-chip variation on clock tree paths with 300ps. The right-hand side shows exactly the same netlist after clock skew scheduling and clock tree synthesis. The slacks have been obtained with a full timing analysis as used for signoff, also taking on-chip variation into account. All negative slacks have disappeared. In this case we improved the frequency of the most critical clock domain by 27%. The corresponding clock tree is shown in Figure 12. It runs at 1.033 Gigahertz [18]. Next we explain how BonnClock constructs such a clock tree, using the input of clock skew scheduling by BonnCycleOpt.

C. Clock Tree Synthesis

The input to BonnClock is a set of sinks, a time interval for each sink, a set of possible sources, a logically correct clock tree serving these sinks, a library of inverters and other books that can be used in the clock tree, and a few parameters, most importantly a slew target. The goal is to replace the initial

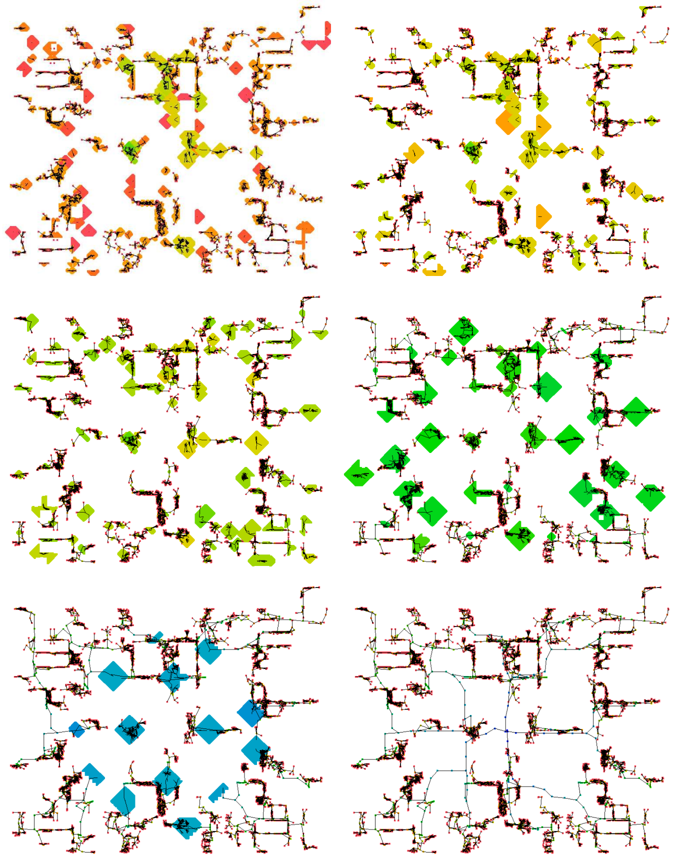

Fig. 10. Different stages of a clock tree construction using BonnClock. The colored octagons indicate areas in which inverters (current sinks) can be placed. The colors correspond to arrival times within the clock tree: blue for signals close to the source, and green, yellow, and red for later arrival times. During the bottom-up construction the octagons slowly converge to the source, here located approximately at the center of the chip.

tree by a logically equivalent clock tree which ensures that all clock signals arrive within the specified time intervals.

First, the input tree is condensed to a minimal tree by identifying equivalent books and removing buffers and inverter pairs. For simplicity we will assume here that the tree contains no special logic and can be constructed with inverters only.

Next we do some preprocessing to determine the approximate distance to a source from every point on the chip, taking into account that some macros can prevent us from going straight towards a source.

BonnClock then proceeds in a bottom-up fashion (cf. Figure 10). Consider a sink s whose earliest feasible arrival time is latest, and consider all sinks whose arrival time intervals contain this point in time. Then we want to find a set of inverters that drives at least s but maybe also some of the other sinks. For each inverter we have a maximum capacitance which it can drive, and the goal is to minimize power consumption.

The input pins of the newly inserted inverters become new sinks, while the sinks driven by them are removed from the current set of sinks. When we insert an inverter, we fix neither its position nor its size. Rather we compute a set of octagons as feasible positions by taking all points with a certain maximal distance from the intersection of the sets of positions of its

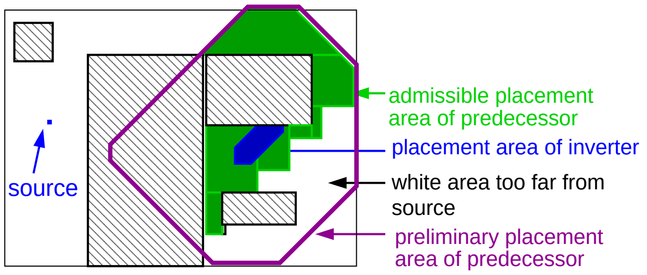
admissible placement area of predecessor
placement area of inverter
white area too far from source
preliminary placement area of predecessor
source

Fig. 11. Computation of the feasible area for a predecessor of an inverter. From all points that are not too far away from the placement area of the inverter (blue) we subtract unusable areas (e.g., those blocked by macros) and points that are too far away from the source. The result (green) can again be represented as a union of octagons.

successors, and subtracting blocked areas and all points that are too far away from a source (cf. Figure 11).

The inverter sizes are determined only at the very end after constructing the complete tree. During the construction we work with solution candidates. A solution candidate is associated with an inverter size, an input slew, a feasible arrival time interval for the input, and a solution candidate for each successor. We prune dominated candidates, i.e. those for which another candidate with the same input slew exists whose time interval contains the time interval of the former. Thus the time intervals imply a natural order of the solution candidates with a given input slew.

Given the set of solution candidates for each successor, we compute a set of solution candidates for a newly inserted inverter as follows. For each input slew at the successors we simultaneously scan the corresponding candidate lists in the natural order and choose maximal intersections of these time intervals. For such a non-dominated candidate set we try all inverter sizes and a discrete set of input slews and check whether they fit. If so, a new candidate is generated.

After an inverter is inserted but before its solution candidates are generated, the successors are placed at a final legal position. It may be necessary to move other objects, but with BonnPlace legalization (cf. Section II-D) we can usually avoid moves with a large impact on timing. There are some other features which pull sinks towards sources, and which cause sinks that are ends of critical paths to be joined early in order to bound negative timing effects due to on-chip variation.

The inverter sizes are selected at the very end by choosing a solution candidate at the root. The best candidate (i.e. the best overall solution) with respect to timing and power consumption is chosen. Due to discretizing slews, assuming bounded RC delays, and legalization, the timing targets may be missed by a small amount, in the order of 20ps. But this impacts the overall timing result only if the deviation occurs in opposite directions at the ends of a critical path.

The overall power consumption is dominated by the bottom stage, where 80–90% of the power is consumed. Therefore the first clustering is very important.

The basic mathematical problem that we face here can be formulated as follows: Given a set $\mathcal{D}$ of sinks, input capacitances $d : \mathcal{D} \to \mathbb{R}_+$, a basic cost $f \in \mathbb{R}_+$ for inserting an inverter, and a capacitance limit $u \in \mathbb{R}_+$, the task is to

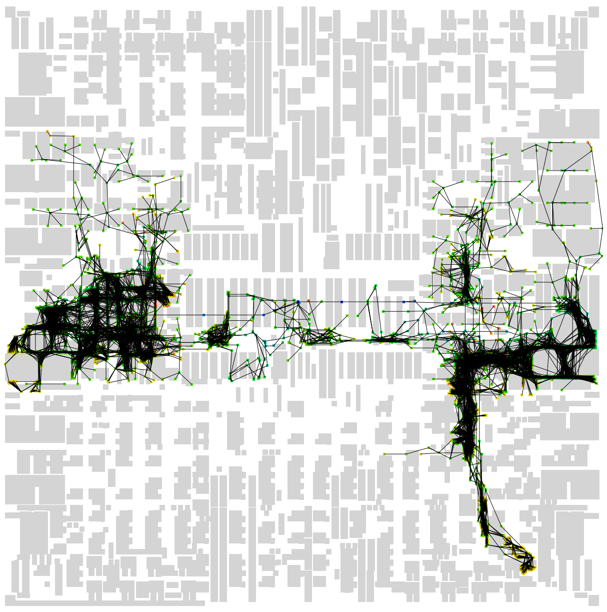

Fig. 12. Gigahertz clock tree built by BonnClock based on the result of BonnCycleOpt shown in Figure 9. Colors indicate different arrival times as in Figure 10. Each net is represented by a star connecting the source to all sinks.

find a partition $\mathcal{D} = D_1 \dot\cup \cdots \dot\cup D_k$ and Steiner trees T_i for D_i ($i = 1, \ldots, k$) with $c(E(T_i)) + d(D_i) \le u$ for $i = 1, \ldots, k$ such that $\sum_{i=1}^{k} c(E(T_i)) + kf$ is minimum.

The first constant-factor approximation algorithm for this problem was given in [27]. It computes a minimum spanning tree on the sinks, removes expensive edges, and splits overloaded connected components. It runs in $O(n \log n)$ time for n sinks and yields excellent results. We combine it with a greedy augmentation approach [18] when there are many non-matching arrival time intervals, and with an exchange and merge heuristic for postoptimization.

By exploiting the time intervals, which are single points only for the few most critical storage elements, and by using an algorithm with provable performance guarantee we can reduce the power consumption substantially.

V. Routing

Due to the enormous instance sizes, most routers including BonnRoute consist of at least two major parts, global and detailed routing. Global routing defines an area for each net to which the search for actual wires in detailed routing is restricted. As global routing works on a much smaller graph, we can globally optimize the most important design objectives. Moreover, global routing has another important function: decide for each placement whether a feasible routing exists and if not, give a certificate of infeasibility.

BonnRoute does not contain any step between global and detailed routing, in particular no track assignment. Track assignment can save running time of the local router for very easy chips. For complex and dense chips track assignment often requires numerous rip-up-and-reroute efforts, which give rise to a substantial increase of the total running time. Due to

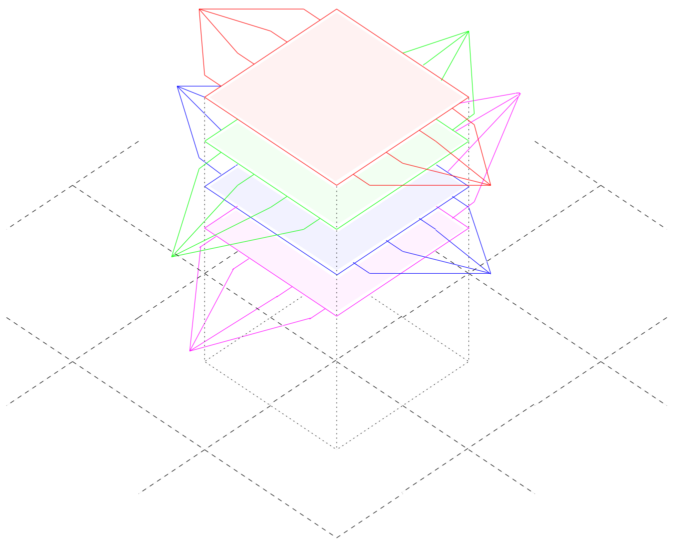

Fig. 13. An instance of the edge-disjoint paths problem for estimating global routing capacities. Dashed lines bound global routing regions. Here we show four wiring planes, each with a commodity (shown in different colors), in alternating preference directions.

our accurate capacity estimation and very fast shortest path algorithm in detailed routing, we do not need any hint other than that provided by global routing.

A. The Global Routing Graph

The global router works on a three-dimensional grid graph which is obtained – as usual – by partitioning the chip area into regions. For classical Manhattan routing this can be done by an axis-parallel grid. In any case, these regions are the vertices of the global routing graph. Adjacent regions are joined by an edge, with a capacity value indicating how many wires of unit width can join the two regions.

For each net we consider the regions that contain at least one of its pins. These vertices of the global routing graph have to be connected by a Steiner tree. If a pin consists of shapes in more than one region we may assign it to one of them, say the one which is closest to the center of gravity of the whole net, or by solving a group Steiner tree problem.

The quality of the global routing depends heavily on the capacity of the global routing edges. A rough estimate has to consider blockages and certain resources for nets whose pins lie in one region only. These nets are not considered in global routing. However, they may use global routing capacity. Therefore we route very short nets, which lie in one region or in two adjacent regions, first in the routing flow, i.e. before global routing. They are then viewed as blockages in global routing. Yet these nets may be rerouted later in local routing if necessary.

Routing short nets before global routing makes better capacity estimates possible, but this also requires more sophisticated algorithms than are usually used for this task. We consider a vertex-disjoint paths problem for every set of four adjacent global routing regions, illustrated in Figure 13. There is a commodity for each wiring plane, and we try to find as many paths for each commodity as possible. Each path may use the plane of its commodity in preference direction and adjacent planes in the orthogonal direction.

An upper bound on the total number of such paths can be obtained by considering each commodity independently and solving a maximum flow problem. However, this is too optimistic and too slow. Instead we compute a set of vertex-disjoint paths (i.e., a lower bound) by a new and very fast multicommodity flow heuristic [29]. It is essentially an augmenting path algorithm but exploits the special structure of a grid graph. For each augmenting path it requires only $O(k)$ constant-time bit pattern operations, where k is the number of edges orthogonal to the preferred wiring direction in the respective layer. In practice, k is less than three for most paths.

This very fast heuristic finds a number of edge-disjoint paths in the region of 90% of the (weak) max-flow upper bound. For a complete chip with about one billion paths it needs 5 minutes of computing time whereas a complete max-flow computation with our implementation of the Goldberg-Tarjan algorithm would need more than a week.

Please note that this algorithm is used only for a better capacity estimation, i.e. for generating accurate input to the main global routing algorithm. However, this better capacity estimate yields much better global routing solutions and allows the detailed router to realize these solutions.

B. Global Routing

In its simplest version, the global routing problem amounts to packing Steiner trees in a graph with edge capacities. A fractional relaxation of this problem can be efficiently solved by an extension of methods for the multicommodity flow problem. However, the approach does not consider today's main design objectives which are timing, signal integrity, power consumption, and manufacturing yield. Minimizing the total length of all Steiner trees is no longer important. Instead, minimizing a weighted sum of the capacitances of all Steiner trees, which is equivalent to minimizing power consumption, is an important objective. Delays on critical paths also depend on the capacitances of their nets. Wire capacitances can no longer be assumed to be proportional to the length, since coupling between neighboring wires plays an increasingly important role. Small detours of nets are often better than the densest possible packing. Spreading wires can also improve the yield.

Our global router is the first algorithm with a provable performance guarantee which takes timing, coupling, yield, and power consumption into account directly. Our global routing algorithm extends earlier work on multicommodity flows, fractional global routing, and randomized rounding.

Let G be the global routing graph, with edge capacities $u : E(G) \to \mathbb{R}$ and lengths $l : E(G) \to \mathbb{R}_+$. Let $\mathcal{N}$ be the set of nets. For each $N \in \mathcal{N}$ we have a set $\mathcal{Y}_N$ of feasible Steiner trees. The set $\mathcal{Y}_N$ may contain all Elmore-delay-optimal Steiner trees of N or, in many cases, it may contain all possible Steiner trees for N in G. Actually, we do not need to know the set $\mathcal{Y}_N$ explicitly. The only assumption which we make is that for each $N \in \mathcal{N}$ and any $\psi : E(G) \to \mathbb{R}_+$ we can find a Steiner tree $Y \in \mathcal{Y}_N$ with $\sum_{e \in E(Y)} \psi(e)$ (almost) minimum sufficiently fast. This assumption is justified since in practical instances almost all nets have less than, say, 10 pins and thus a dynamic programming algorithm for finding an optimum Steiner tree is very fast. With $w(N, e) \in \mathbb{R}_+$ we denote the width of net N at edge e. A straightforward integer programming formulation of the global routing problem is:

$$\begin{array}{lll}
\min & \displaystyle\sum_{N\in\mathcal{N}} \sum_{e\in E(G)} l(e) \sum_{Y\in\mathcal{Y}_N | e\in E(Y)} x_{N,Y} & \\
\text{s.t.} & \displaystyle\sum_{N\in\mathcal{N}} \sum_{Y\in\mathcal{Y}_N : e\in E(Y)} w(N,e) x_{N,Y} \le u(e) & (e \in E(G)) \\
& \displaystyle\sum_{Y\in\mathcal{Y}_N} x_{N,Y} = 1 & (N \in \mathcal{N}) \\
& x_{N,Y} \in \{0,1\} & (N \in \mathcal{N}, Y \in \mathcal{Y}_N)
\end{array}$$

Here the decision variable $x_{N,Y}$ is 1 iff the Steiner tree Y is chosen for net N. The decision whether this integer programming problem has a feasible solution is already NP-complete. Thus, we relax the problem by assuming $x_{N,Y} \in [0, 1]$. Raghavan and Thompson [33], [34] proposed solving the LP relaxation first, and then using randomized rounding to obtain an integral solution whose maximum capacity violation can be bounded. Although the LP relaxation has exponentially many variables, it can be solved in practice for moderate instance sizes since it has only $|E(G)|+|\mathcal{N}|$ many constraints. Therefore all but $|E(G)| + |\mathcal{N}|$ variables are zero in an optimum solution. However, for current complex chips with millions of nets and edges, all exact algorithms for solving the LP relaxation are far too slow.

Fortunately, there exist combinatorial fully polynomial approximation schemes, i.e. algorithms that compute a feasible solution of the LP relaxation which is within a factor of $1 + \epsilon$ of the optimum, and whose running time is bounded by a polynomial in $|V(G)|$ and $\frac{1}{\epsilon}$, for any accuracy $\epsilon > 0$. If each net has exactly two pins, $\mathcal{Y}_N$ contains all possible paths connecting N, and $w \equiv 1$, the global routing problem reduces to the edge-disjoint paths problem whose fractional relaxation is the multicommodity flow problem. Shahrokhi and Matula [41] have developed this first fully polynomial approximation scheme for multicommodity flows. Carden, Li and Cheng [12] first applied this approach to global routing, while Albrecht [1] applied a modification of the approximation algorithm by Garg and Könemann [16]. However, these approaches did not consider the above-mentioned design objectives, like timing, power, and yield.

The power consumption of a chip induced by its wires is proportional to the weighted sum of all capacitances, weighted by switching activities. The capacitance of a net consists of area capacitance, proportional to length times width, fringing capacitance, proportional to length, and coupling capacitance, proportional to length if adjacent wires exist. The coupling capacitance also depends on the distance between adjacent wires. In older technologies coupling capacitances were quite small and therefore could be ignored. In deep submicron technologies coupling matters a lot.

Since the width $w(e, N)$ of a wire of net N at edge e is known, we also know the maximum capacitance of this wire under the assumption that parallel wires run at both sides with minimum distance. We denote this maximum capacitance by $l(e, N)$. We further assume that extra space $s(e, N)$ assigned to a wire reduces the coupling capacitance by $v(e, N)$, and for less extra space the capacitance reduction is linear. This means that the space $w(N, e)+y(e, N)s(e, N)$ with $0 \le y(e, N) \le 1$ results in a capacitance $l(e, N) - y(e, N)v(e, N)$. This is – of course – a simplification, since coupling does not depend linearly on distance and also blockages, pin shapes and vias are ignored. Yet, quite accurate results can be obtained by this simple model.

Similarly to minimizing power consumption based on the above capacitance model, we can optimize yield by replacing capacitance by "critical area", i.e. the sensitivity of a layout to random defects [30].

Moreover, we can also consider timing restrictions. This can be done by excluding from the set $\mathcal{Y}_N$ all Steiner trees with large detours, or by imposing upper bounds on the weighted sums of capacitances of nets that belong to critical paths. For this purpose, we first do a static timing analysis under the assumption that every net has some expected capacitance. The set $\mathcal{Y}_N$ will contain only Steiner trees with capacitance below this expected value. We enumerate all paths which have negative slacks under this assumption. We compute the sensitivity of the nets of negative slack paths to capacitance changes, and use these values to translate the delay bound to appropriate bounds on the weighted sum of capacitances for each path. To compute reasonable expected capacitances we can apply weighted slack balancing (cf. Section IV-B) using delay sensitivity and congestion information. Altogether we get a family $\mathcal{M}$ of subsets of $\mathcal{N}$ with $\mathcal{N} \in \mathcal{M}$ and bounds $U : \mathcal{M} \to \mathbb{R}_+$ and weights $c(M, N) \in \mathbb{R}_+$ for $N \in M \in \mathcal{M}$.

With these additional assumptions and this notation we can generalize the original integer programming formulation of the global routing problem to:

$$
\begin{aligned}
&\min\ \lambda \\
&\text{subject to} \\
&\sum_{Y \in \mathcal{Y}_N} x_{N,Y} = 1 && (N \in \mathcal{N}) \\
&\sum_{N \in M} c(M,N) \left(\sum_{Y \in \mathcal{Y}_N} \sum_{e \in E(Y)} l(e,N) x_{N,Y} - \sum_{e \in E(G)} v(e,N) y_{e,N} \right) \le \lambda U(M) && (M \in \mathcal{M}) \\
&\sum_{N \in \mathcal{N}} \left(\sum_{Y \in \mathcal{Y}_N : e \in E(Y)} w(N,e) x_{N,Y} + s(e,N) y_{e,N} \right) \le \lambda u(e) && (e \in E(G)) \\
&y_{e,N} \le \sum_{Y \in \mathcal{Y}_N : e \in E(Y)} x_{N,Y} && (e \in E(G), N \in \mathcal{N}) \\
&y_{e,N} \ge 0 && (e \in E(G), N \in \mathcal{N}) \\
&x_{N,Y} \in \{0,1\} && (N \in \mathcal{N}, Y \in \mathcal{Y}_N)
\end{aligned}
$$

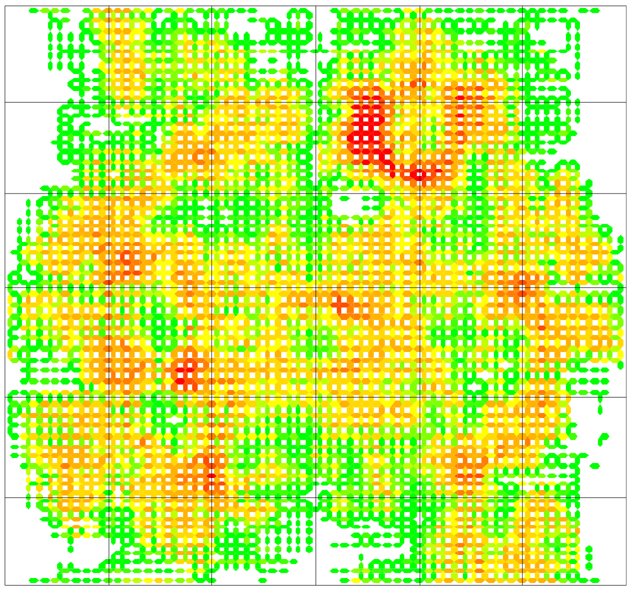
Fig. 14. A typical global routing congestion map. Each edge corresponds to approximately 100 global routing edges (and to approximately 10000 detailed routing channels). Red, orange, yellow, green, and white edges correspond to an average load of approximately 90–100%, 70–90%, 60–70%, 40–60%, and less than 40%.

Now we again relax this integer program to a linear program. In [44] we developed a fully polynomial approximation scheme for this LP and its dual. The algorithm always gives a fractional dual solution and therefore a certificate of infeasibility if a given placement is not routable. We also showed how to make randomized rounding work [44].

This approach is quite general. It allows us to add further linear constraints to the classical fractional primal-dual formulation of the multicommodity flow problem. Here we have modeled timing, yield, and power consumption, but we may think of other constraints if further technological or design restrictions come up.

Figure 14 shows a typical result of global routing. In the dense (red and orange) areas the main challenge is to find a feasible solution, while in other areas there is room for optimizing objectives like power or yield. Experimental results show a significant improvement over previous approaches which optimized netlength and number of vias, both in terms of power consumption and expected manufacturing yield [30].

C. Detailed Routing

The task of detailed routing is to determine the exact layout of the metal realizations of the nets. We need an efficient data structure that stores all metal shapes and allows fast queries. Grid-based routers define routing tracks (and minimum distance) and work with a detailed routing graph G which is an incomplete three-dimensional grid graph, i.e. $V(G) \subseteq \{x_{\min}, \ldots, x_{\max}\} \times \{y_{\min}, \ldots, y_{\max}\} \times \{1, \ldots, z_{\max}\}$ and $((x,y,z),(x',y',z')) \in E(G)$ only if $|x - x'| + |y - y'| + |z - z'| = 1$.

The z-coordinate models the different routing layers of the chip and $z_{\max}$ is typically around 10. We can assume without loss of generality that the x- and y-coordinates correspond to the routing tracks; typically the number of routing tracks in each plane, i.e. $x_{\max} - x_{\min}$ and $y_{\max} - y_{\min}$, is in the order of magnitude of 10^5, resulting in a graph with approximately 10^{11} vertices. The graph is incomplete because some parts are reserved for internal circuit structures or power supply, and some nets may have been routed earlier.

To find millions of vertex-disjoint Steiner trees in such a huge graph is very challenging. Thus we decompose this task, route the nets and even the two-point connections making up the Steiner tree for each net sequentially. Then the elementary algorithmic task is to determine shortest paths within the detailed routing graph (or within a part of it, as specified by global routing).

Whereas the computation of shortest paths is probably the most basic and well-studied algorithmic problem of discrete mathematics [26], the size of G and the number of shortest paths that have to be found concurrently makes the use of textbook versions of shortest path algorithms impossible. The basic algorithm for finding a shortest path connecting two given vertices in a digraph with nonnegative arc weights is Dijkstra's algorithm. Its theoretically fastest implementation, with Fibonacci heaps, runs in $O(m + n \log n)$ time, where n and m denote the number of vertices and edges, respectively [15]. For our purposes this is much too slow. We therefore apply various strategies to speed up Dijkstra's algorithm.

Since we are not just looking for one path but have to embed millions of disjoint trees, the information provided by global routing is most important. For each two-point connection global routing determines a corridor essentially consisting of the global routing tiles to which this net was assigned in global routing. If we find a shortest path for the two-point connection within this corridor, the capacity estimates used during global routing approximately guarantee that all desired paths can be realized disjointly. Furthermore, we get a dramatic speedup by restricting the path search to this corridor, which usually represents a very small fraction of the entire routing graph.

The second important factor speeding up our shortest path algorithm is the way in which we store the distance information. Whereas Dijkstra's algorithm labels individual vertices, we consider intervals of consecutive vertices that are similar with respect to their usability and their distance properties. Since the layers are assigned preferred routing directions, the intervals are chosen parallel to these. By the similarity of the vertices in one interval we mean that their distance properties can be encoded more efficiently than by storing numbers for each individual vertex. If e.g. the distance increases by one unit from vertex to vertex we just need to store the distance information for one vertex and the increment direction. Our version of Dijkstra's algorithm [19] labels intervals instead of vertices, and its time complexity therefore depends on the number of intervals, which is typically about 100 times smaller than the number of vertices. A sophisticated data structure for storing the intervals and answering queries very fast is the basis of this algorithm and also of our efficient shared-memory parallelization.

The last factor speeding up our path search is the use of a future cost estimate, which is a lower bound on the distance

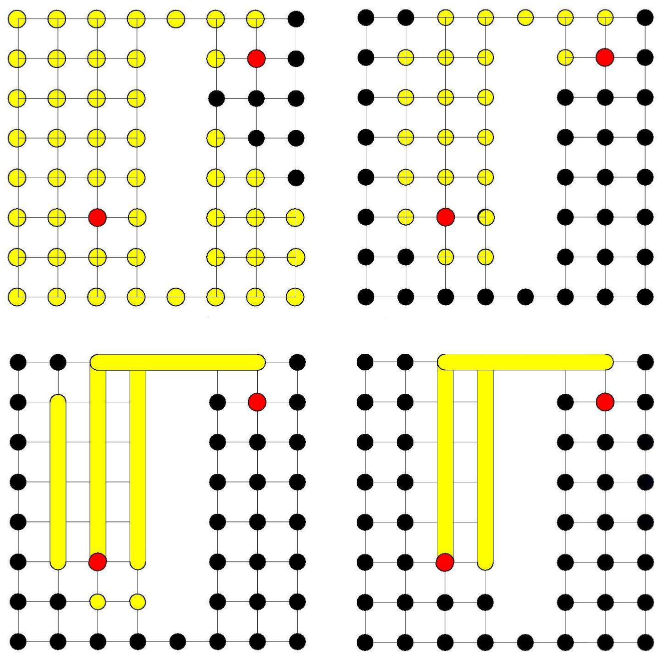

Fig. 15. Dijkstra's algorithm without (left) and with (right) future cost, point-based (top) and interval-based (bottom). We require a shortest path from the red vertex in the bottom left to the red vertex in the upper right part. Points or intervals labeled by Dijkstra's algorithm are shown in yellow. The running time is roughly proportional to the number of labeled points (50 versus 24) or intervals (7 versus 4 in this example).

of vertices to a given target set of vertices. Suppose we are looking for a path from s to t in G with respect to edge weights $c : E(G) \to \mathbb{R}_+$, which reflect higher costs for vias and jogs (wires orthogonal to the preferred direction) and can also be used to find optimal rip-up sets. Let $l(x)$ be a lower bound on the distance from x to t for any vertex $x \in V$. Then we may apply Dijkstra's algorithm to the costs $c'(x, y) := c(\{x, y\}) - l(x) + l(y)$. For any s-t-path P we have $c'(P) = c(P) - l(s) + l(t)$, and hence shortest paths with respect to c' are also shortest paths with respect to c. If l is a good lower bound, i.e. close to the exact distance, and satisfies the natural condition $l(x) \leq c(\{x, y\}) + l(y)$ for all $\{x, y\} \in E(G)$, then this results in a significant speedup. This is illustrated by Figure 15.

If the future cost estimate is exact, our procedure will only label intervals that contain vertices lying on shortest paths.

Clearly, improving the accuracy of the future cost estimate improves the running time of the path search and there is a tradeoff between the time needed to improve the future cost and the time saved during path search. The fastest future cost estimate which already leads to considerable speedup and can be calculated in $O(1)$ time is the ℓ_1-distance. We are currently experimenting [31] with a much more accurate future cost which relies on a preliminary labeling algorithm working on the global routing tiles. Rather than labeling vertices or intervals (=1-dimensional arrays of vertices) it labels 2-dimensional arrays of vertices. The information computed by this (very fast) preliminary labeling algorithm is then used to compute excellent future cost estimates in constant time during the main path search.

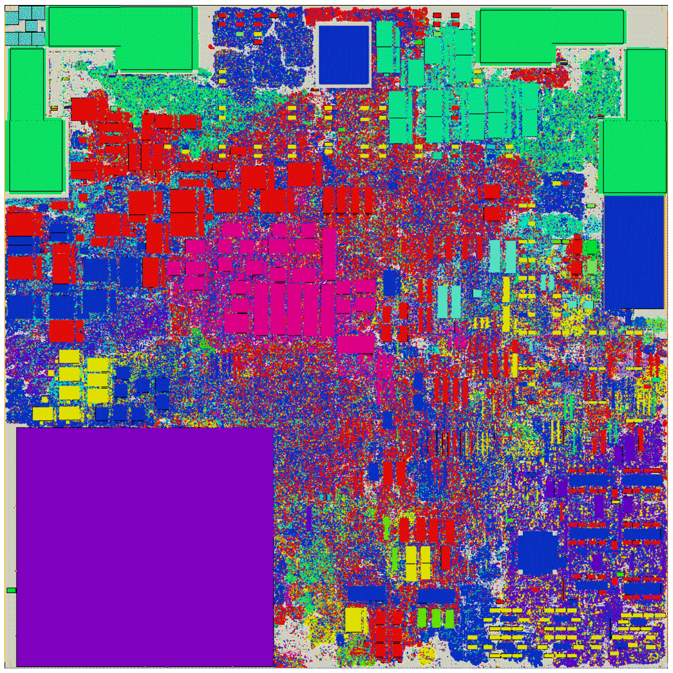

Fig. 16. A system on a chip designed in 2006 with BonnTools. This 90nm design for a forthcoming IBM server has almost 5 million nets on the top level and runs with frequencies up to 1.5GHz.

Finally, we note that the use of the detailed routing graph as the basis of our interval data structure is not restricted to grid-based routing styles. In fact, it does not matter whether wires lie on or off predefined routing tracks. There is only a slight overhead for wires thicker than one track. Our data structure, although behaving in essentially the same way as for grid-based routing, efficiently captures the geometry of arbitrary (gridless) routing shapes. Each shape is associated with the vertex of the detailed routing graph that represents the area containing the shape. The intervals are thus associated with (gridless) routing patterns. Since the total number of different patterns is relatively small, the memory overhead remains acceptable. The running time of the core routines, in particular our implementation of Dijkstra's algorithm, is almost not affected. Thus our algorithmic solutions provide a solid basis for grid-based and gridless libraries.

VI. Conclusion, Outlook

We have demonstrated that mathematics can yield better solutions for leading-edge chips. Several complete microprocessor series (cf., e.g., [14], [23]) and many leading-edge ASICs (cf., e.g., [24], [18]) have been designed with BonnTools. Many additional ones are in the design centers at the time of writing. A very recent example, a chip designed by IBM with BonnTools in 2006, is shown in Figure 16.

On the other hand, chip design is inspiring a great deal of interesting work in mathematics. Indeed, most classical problems in combinatorial optimization, and many new ones, have been applied to chip design. Some algorithms originally developed for VLSI design automation are applied also in other contexts.

However, there remains a lot of work to do. Exponentially increasing instance sizes continue to pose challenges. Even some classical problems (e.g., logic synthesis) have no satisfactory solution yet, and future technologies continuously bring new problems. Yet we strongly believe that mathematics will continue to play a vital role in facing these challenges.

Acknowledgements

We thank all current and former members of our team in Bonn. Moreover, we thank our cooperation partners at IBM and Magma.

References

[1] Albrecht, C.: Global routing by new approximation algorithms for multicommodity flow. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 20 (2001), 622–632

[2] Albrecht, C., Korte, B., Schietke, J., and Vygen, J.: Cycle time and slack optimization for VLSI-chips. Proceedings of the IEEE International Conference on Computer-Aided Design (1999), 232–238

[3] Albrecht, C., Korte, B., Schietke, J., and Vygen, J.: Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip. Discrete Applied Mathematics 123 (2002), 103–127

[4] Bartoschek, C., Held, S., Rautenbach, D., and Vygen, J.: Efficient generation of short and fast repeater tree topologies. Proceedings of the International Symposium on Physical Design (2006), 120–127

[5] Brenner, U.: A faster polynomial algorithm for the unbalanced Hitchcock transportation problem. Report No. 05954, Research Institute for Discrete Mathematics, University of Bonn, 2005

[6] Brenner, U., Pauli, A., and Vygen, J.: Almost optimal placement legalization by minimum cost flow and dynamic programming. Proceedings of the International Symposium on Physical Design (2004), 2–9

[7] Brenner, U., and Rohe, A.: An effective congestion driven placement framework. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 22 (2003), 387–394

[8] Brenner, U., and Struzyna, M.: Faster and better global placement by a new transportation problem. Proceedings of the 42nd IEEE/ACM Design Automation Conference (2005), 591–596

[9] Brenner, U., and Vygen, J.: Faster optimal single-row placement with fixed ordering. Design, Automation and Test in Europe, Proceedings, IEEE 2000, 117–121

[10] Brenner, U., and Vygen, J.: Worst-case ratios of networks in the rectilinear plane. Networks 38 (2001), 126–139

[11] Brenner, U., and Vygen, J.: Legalizing a placement with minimum total movement. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 23 (2004), 1597–1613

[12] Carden IV, R.C., Li, J., and Cheng, C.-K.: A global router with a theoretical bound on the optimum solution. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 15 (1996), 208–216

[13] Chen, C.-P., Chu, C.C.N., and Wong, D.F.: Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18 (1999), 1014–1025

[14] Fassnacht, U., and Schietke, J.: Timing analysis and optimization of a high-performance CMOS processor chipset. Design, Automation and Test in Europe, Proceedings, IEEE 1998, 325–331

[15] Fredman, M.L., and Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization problems. Journal of the ACM 34 (1987), 596–615

[16] Garg, N., and Könemann, J.: Faster and simpler algorithms for multicommodity flow and other fractional packing problems. Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (1998), 300–309

[17] Held, S.: Algorithms for potential balancing problems and applications in VLSI design [in German]. Diploma thesis, University of Bonn, 2001

[18] Held, S., Korte, B., Maßberg, J., Ringe, M., and Vygen, J.: Clock scheduling and clocktree construction for high performance ASICs. Proceedings of the IEEE International Conference on Computer-Aided Design (2003), 232–239

[19] Hetzel, A.: A sequential detailed router for huge grid graphs. Design, Automation and Test in Europe, Proceedings, IEEE 1998, 332–338

[20] Kahng, A.B., Tucker, P., and Zelikovsky, A.: Optimization of linear placements for wirelength minimization with free sites. Proceedings of the Asia and South Pacific Design Automation Conference, 1999, 241–244

[21] Karp, R.M.: A characterization of the minimum mean cycle in a digraph. Discrete Mathematics 23 (1978), 309–311

[22] Kleinhans, J.M., Sigl, G., Johannes, F.M., and Antreich, K.J.: GORDIAN: VLSI placement by quadratic programming and slicing optimization. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10 (1991), 356–365

[23] Koehl, J., Baur, U., Ludwig, T., Kick, B., and Pflueger, T.: A flat, timing-driven design system for a high-performance CMOS processor chipset. Design, Automation, and Test in Europe, Proceedings, IEEE 1998, 312–320

[24] Koehl, J., Lackey, D.E., and Doerre, G.W.: IBM's 50 million gate ASICs. Proceedings of the Asia and South Pacific Design Automation Conference, IEEE 2003, 628–634

[25] Korte, B., Lovász, L., Prömel, H.J., and Schrijver, A. (Eds.): Paths, Flows, and VLSI-Layout. Springer, Berlin 1990

[26] Korte, B., and Vygen, J.: Combinatorial Optimization: Theory and Algorithms. Third edition. Springer, Berlin 2006

[27] Maßberg, J., and Vygen, J.: Approximation algorithms for network design and facility location with service capacities. In: Approximation, Randomization and Combinatorial Optimization; Proceedings of the 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 2005); LNCS 3624 (C. Chekuri, K. Jansen, J.D.P. Rolim, L. Trevisan, eds). Springer, Berlin 2005, pp. 158–169

[28] Minoux, M.: Mathematical Programming: Theory and Algorithms. Wiley, Chichester 1986

[29] Müller, D.: Determining routing capacities in global routing of VLSI chips [in German]. Diploma thesis, University of Bonn, 2002

[30] Müller, D.: Optimizing yield in global routing. Proceedings of the IEEE International Conference on Computer-Aided Design (2006), to appear

[31] Peyer, S., Rautenbach, D., and Vygen, J.: Generalizing Dijkstra's algorithm for shortest paths in huge graphs, with applications to VLSI routing. Manuscript 2006.

[32] Philips, S., and Dessouky, M.: Solving the project time/cost tradeoff problem using the minimal cut concept. Management Science 24 (1977), 393–400

[33] Raghavan, P., and Thompson, C.D.: Randomized rounding: a technique for provably good algorithms and algorithmic proofs. Combinatorica 7 (1987), 365–374

[34] Raghavan, P., and Thompson, C.D.: Multiterminal global routing: a deterministic approximation. Algorithmica 6 (1991), 73–82

[35] Rautenbach, D., Szegedy, C., and Werber, J.: Delay optimization of linear depth Boolean circuits with prescribed input arrival times. Journal of Discrete Algorithms, to appear

[36] Rautenbach, D., Szegedy, C., and Werber, J.: Fast circuits for functions whose inputs have specified arrival times. Report No. 03933, Research Institute for Discrete Mathematics, University of Bonn, 2003

[37] Rautenbach, D., Szegedy, C., and Werber, J.: Asymptotically optimal Boolean circuits for functions of the form $g_{n-1}(g_{n-2}(...g_3(g_2(g_1(x_1,x_2),x_3),x_4)...,x_{n-1}),x_n)$. Report No. 03931, Research Institute for Discrete Mathematics, University of Bonn, 2003

[38] Rautenbach, D., and Szegedy, C.: A class of problems for which cyclic relaxation converges linearly. Report No. 04939, Research Institute for Discrete Mathematics, University of Bonn, 2004

[39] Rautenbach, D., and Szegedy, C.: A subgradient method using alternating projections. Report No. 04940, Research Institute for Discrete Mathematics, University of Bonn, 2004

[40] Schneider, H., and Schneider, M.H.: Max-balancing weighted directed graphs and matrix scaling. Mathematics of Operations Research 16 (1991), 208–222

[41] Shahrokhi, F., and Matula, D.W.: The maximum concurrent flow problem. Journal of the ACM 37 (1990), 318–334

[42] Vygen, J.: Algorithms for large-scale flat placement. Proceedings of the 34th IEEE/ACM Design Automation Conference (1997), 746–751

[43] Vygen, J.: Algorithms for detailed placement of standard cells. Design, Automation and Test in Europe, Proceedings, IEEE 1998, 321–324

[44] Vygen, J.: Near-optimum global routing with coupling, delay bounds, and power consumption. In: Integer Programming and Combinatorial Optimization; Proceedings of the 10th International IPCO Conference; LNCS 3064 (G. Nemhauser, D. Bienstock, eds.), Springer, Berlin 2004, pp. 308–324

[45] Vygen, J.: Geometric quadrisection in linear time, with application to VLSI placement. Discrete Optimization 2 (2005), 362–390

[46] Vygen, J.: Slack in static timing analysis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25 (2006), 1876–1885

[47] Vygen, J.: New theoretical results on quadratic placement. Integration, the VLSI Journal, to appear

[48] Young, N.E., Tarjan, R.E., and Orlin, J.B.: Faster parametric shortest path and minimum balance algorithms. Networks 21 (1991), 205–221