A Network-Flow Approach to Timing-Driven Incremental Placement for ASICs *

Shantanu Dutt, Huan Ren, Fenghua Yuan and Vishal Suthar
Dept. of ECE, University of Illinois-Chicago
dutt@ece.uic.edu, hren2@uic.edu

ABSTRACT

We present a novel incremental placement methodology called FlowPlace for significantly reducing critical path delays of placed standard-cell circuits. FlowPlace includes: a) a timing-driven (TD) analytical global placer TAN that uses accurate delay functions and minimizes a combination of linear and quadratic objective functions; b) a network flow based detailed placer TIF that has new and effective techniques for performing TD incremental placement and satisfying row-length (white space) constraints. We have obtained results on three sets of benchmarks: i) TD versions of the ibm benchmark suite that we have constructed; ii) benchmarks used in TD-Dragon; iii) the Faraday benchmarks. Results show that starting with Dragon-placed circuits, we are able to obtain up to 34% and an average of 18% improvement in critical path delays, at an average of 17.5% of the run-time of the Dragon placer. Starting with a state-of-the-art TD placer TD-Dragon, for the TD-Dragon benchmarks we obtain up to about 10% and an average of 4.3% delay improvement with 12% of TD-Dragon's run times; this is significant as we are extracting performance improvements from a performance-optimized layout. Wire length deterioration on the average over all benchmark suites is less than 8%.

1. INTRODUCTION

Due to the increasing ratio of interconnect to gate delays in very deep submicron (VDSM) designs, and the large impact that placement plays on the final wire length (WL) as well as performance, WL and timing consideration during placement is critical. Timing driven (TD) placement algorithms can be divided into 3 categories. 1) partition-based, like [9, 12], 2) simulated annealing (SA) based, like [11, 15], and 3) analytical [8]. Circuit timing optimization is basically a path-based problem, though it is impractical to track delays of all paths, since their numbers are generally exponential in circuit size [8]. Hence, timing constraints on paths are usually converted to either net/edge weights or constraints such as net delay bounds, yielding more tractable *net-based* methods. In a recent work [8], a novel edge weight function was proposed that, together with its new objective function, solves the convergence problem in net-based methods—delay reductions along critical paths are sometimes obtained at the expense of delay increases in non-critical paths, to the extent that the circuit delay reduces little, if at all. In [15], a SA approach is used along with delay bounds on nets. The slack assignment approach in the paper ensures that estimated long nets are assigned a larger delay bound so that they are not be overly constrained. The objective function is to minimize the sum of delay violations across all nets.

Another approach to TD placement is via targeted *incremental placement*. On an initial base placement, an incremental TD placer can focus on reducing delays of the most critical paths. This will greatly reduce the number of paths that need to be considered. Also, more timing information can be derived if there is an initial placement; thus delay and slack estimates, and thereby cost functions, are more accurate. Furthermore, by its very nature, TD incremental placement, if done properly, implicitly solves the aforementioned convergence problem, since it minimizes placement changes to the non-critical paths, thereby limiting any delay increases in them. TD incremental placement also finds applications in ECO scenarios where changes in stages above the physical design (PD) level generally percolate down to required changes in the placement and routing stages. In such applications, TD incremental placement would make the required placement changes, while minimizing placement changes in the unaffected portion of the circuit, and minimizing any deterioration in critical path delays. TD incremental placement can also be invoked in ECOs for the express purpose of reducing delays of paths that violate target clock speed constraints via appropriate placement changes in cells on these paths. It is in the context of the first and third applications that we will describe our TD incremental placer, though it can also be used in the general ECO context.

A TD incremental placer was proposed in [14] that directly controls the delay of critical or near-critical paths. It explicitly sets delay constraints for all the critical paths based on the half-perimeter bounding box (HPBB) net lengths on these paths. It then finds a solution to these constraints while minimizing total HPBB WL change in the circuit using linear programming. This method only takes BB length into consideration, which is only one component of sink delays in a net, resulting in less than highly-accurate timing estimates. Also, non-critical nets are ignored, and thus the convergence problem mentioned earlier can surface.

We propose a timing-driven incremental placer FlowPlace

*This work was supported by NSF grant CCR-0204097. We also gratefully acknowledge the permission of Artisan Components, Inc. for the use of the cell-timing libraries of the TD-Dragon benchmarks, and also the help of the TD-Dragon authors in this regard.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICCAD'06 November 5-9, 2006, San Jose, CA
Copyright 2006 ACM 1-59593-389-1/06/0011 ...$5.00.

that addresses many of the above issues. It has two major components. First, an incremental TD analytical placer TAN is used to find an initial placement, possibly with overlaps. Then a TD detailed placer TIF is used to get a legal placement that minimizes critical path delay increase over that of TAN's placement. Our TD analytical placer extends the basic techniques of Gordian [10] and Gordian-L [13] to optimize a TD objective function with quadratic as well as linear terms, and also has carefully designed objective and weight functions. The detailed placement algorithm uses a network flow based method. Network flow has been used previously for solving the legalization problem in standard-cell circuits [4, 5]. In both works, the network flow modeling is similar to ours on some high-level issues: cells are represented by nodes and possible movement of cells are represented by arcs from them to destination positions. The properties of network flow were used in these works to remove cell overlaps, and minimize the sum of flow costs while doing so. However, their objectives were to minimize total WL which can be more easily modeled by sum of flow costs. Our objective is to minimize the delay of critical paths rather than the sum of delays. To this end, we use more complex cost functions and flow graph structures to make sure that the sum of flow costs is a good indicator of mainly critical-path delay changes.

The rest of the paper is organized as follows. Section 2 discusses some basic issues about incremental TD placement and the high-level flow of our methodology. In Sec. 3 we present various aspects of our TD analytical placer including its objective function and accurate interconnect delay estimates. In Sec. 4, the network-flow based TD incremental detailed placer is discussed at length. Section 5 presents experimental results and we conclude in Sec. 6.

2. TD INCREMENTAL PLACEMENT AND METHODOLOGY FLOW

The TD incremental placement problem can be formally stated as follows.

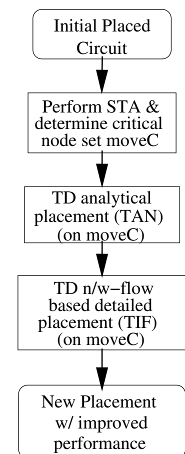

Figure 1: The flowchart of our TD incremental placer FlowPlace.

Input: A placed circuit $\mathcal{PC}$ with some vacant positions both cell movements and deterioration of placement metrics like total wirelength (WL) and chip area, and (2) either: (a) the critical path delay in $\mathcal{PC}'$ is not increased beyond the one in $\mathcal{PC}$ (this applies in applications where the target clock speed has been met, and the ECO process is used to rectify other circuit problems), or (b) the critical path in $\mathcal{PC}'$ is significantly improved compared to the one in the previous layout—we will focus on this objective in this paper, though our incremental placer can also be used to tackle applications of type 2(a) as well.

Fig. 1 shows the flow of our TD incremental placer used in applications of type (b) given above. We start from a placed circuit, and identify all critical and near-critical paths using static timing analysis (STA). Let this set of paths be $\mathcal{P}$. After $\mathcal{P}$ is identified, we remove either: (i) only the cells in $\mathcal{P}$ from the layout, or (ii) all cells in all nets in $\mathcal{P}$ from the layout. The removed cells form the cell set $moveC$ (nets connected to cells in $moveC$ are denoted by $moveN$) that will be replaced by our TD incremental placer with the goal of reducing the critical path delay. This is achieved by a combination of a TD analytical placer TAN in which $moveC$ constitutes the set of movable cells and the minimization function is a sum of net-delay functions weighed inversely by their path slacks, that have both linear and a quadratic interconnect-length terms. Our main contribution in this part is two-fold. The first is developing an accurate and detailed pre-routing net-delay function, and determining net weights so that the net-delays of critical paths have the highest minimization priority. The second is performing both quadratic and linear optimization simultaneously.

The output of our TD analytical placer will generally be an illegal placement for cells in $moveC$—the cell positions determined will generally not be in cell rows and may overlap each other or cells in $\mathcal{PC}$. However, these cell positions provides starting points for our detailed TD placer that uses a novel network flow method for placing new cells in legal positions and moving existing cells minimally to accommodate this in such a way that the critical path delay is optimized and the row-length (i.e., row white space) constraint is satisfied. This TD min-cost max-flow white-space satisfying algorithm, called TIF, is the major contribution of this paper. The basic problem of TD incremental placement (and placement in general) is at its core a constraint-satisfying discrete optimization problem (DOP). By using a network flow approach to solve it, we are using a continuous optimization approach, and thus certain "illegalities" are introduced in the solution for the core problem. We thus also describe in Sec. 4 the in-processing methods we use for: a) legalizing the continuous solution of the network flow process, and b) satisfying white-space constraints that are not completely modeled by standard capacity constraints in the network flow graph.

2.1 STA and Path Slacks

We perform STA to determine delays to the output pins or flip-flops (FFs) of the circuit; each of these "terminal" pins have a max-delay path to them, and the maximum delay over all these paths is the *critical path delay*. We define a *near-critical path* as a max-delay path to a terminal pin whose delay is within a $(1-\epsilon)$ fraction of the critical path delay; we use $\epsilon = 0.1$ in our experiments. A path P's *slack* $S(P)$ is defined as the difference between the *required arrival time (RAT)* at the terminal pin of P and the *arrival time (AT)* of P. We assume a single target clock speed and thus uniform RATs at all terminal pins (our methods easily apply to non-uniform RATs as well). For the purpose of meaningful slack-driven cost functions to minimize critical interconnect lengths, we need positive slacks, and we thus bootstrap our methods by defining the RAT of the terminal pin of the critical path as $(1+\alpha)$ times the critical path delay; we use $\alpha = 0.1$ in our experiments. This ensures positive slack for all paths, and of course smaller slacks for more critical paths.

3. TD ANALYTICAL GLOBAL PLACEMENT

Our analytical placer TAN is a TD extension of a combination of Gordian [10] and Gordian-L [13]—we optimize an objective function that contains both linear and quadratic terms.

3.1 Basic Gordian and Gordian-L

Gordian [10] is a quadratic programming technique for cell placement for quadratic WL minimization. The quadratic net

length estimate can be based on either a clique or a star-graph model. For the latter (see Fig. 2(a)), which we use in our TD objective function, the quadratic net length of net n_j with k pins is given by:

$$L^2(n_j) = \sum_{u_i \in n_j} (x_i - x_c)^2 + (y_i - y_c)^2$$

where (x_i, y_i) are the coordinates of pin u_i, (x_c, y_c) is the coordinate of the centroid of the pins of n_j, with $x_c\ (y_c) = (1/k) \times \sum_{u_i \in n_j} x_i\ (y_i)$.

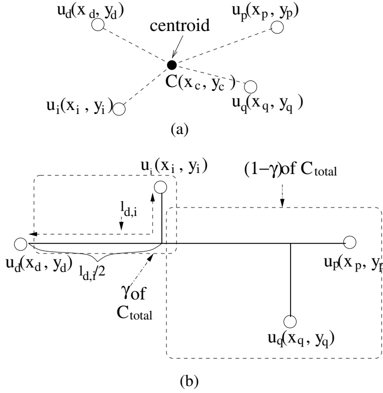

Figure 2: (a) The *star-graph* model for net length estimate. (b) Interconnect delay computation in a pre-routing placement. C_{total} is the total (net and load) capacitance seen by the driver.

For a circuit netlist G, Gordian performs an optimization of the quadratic objective function $\sum_{n_j \in nets(G)} L^2(n_j)$. The linear constraints include those on the coordinates to be initially within chip boundaries, and then within boundaries of subregions—after each Gordian phase in a region, the cells are partitioned, based on their solution coordinates, into two subregions by a cutline perpendicular to the optimization dimension (x or y). This prevents cell overlaps among the two groups and ultimately between every subgroup of cells where this hierarchical process ends.

Gordian-L [13] applies an additional inner-iteration for the optimization in each subregion, which essentially comprises of dividing in the $(m+1)$'th inner iteration, each $L^2(n_j)$ part of the objective function by a net-centric linear-length quantity given by $\eta_j^m = \sum_{u_i \in n_j} |x_i^m - x_c^m|$ (for the optimization along the horizontal dimension), where x_i^m is the value of the x-coordinate of u_i after the m'th iteration, and $\eta_i^0 = 1$. This has the effect of linearizing the objective function at the end of the inner iteration.

3.2 Net Delays and Objective Function

We assume that we start with an unrouted placement [1], and thus use the routing model shown in Fig. 2(b). For a net n_j with driver u_d, and $k-1 \geq 1$ sinks, let R_d be the driving resistance, C_g the load capacitance of a sink pin[2], r (c) the unit wire resistance (capacitance), and $l_{d,i}$ the interconnect length connecting driver u_d to sink u_i; see Fig. 2(b). Referring to this figure and considering a sink u_i in n_j, the delay $D(u_i, n_j)$ to it (using the Elmore delay model) from the driver u_d, consists of three parts:

$$D_1(n_j) = R_d(c \cdot L(n_j) + (k-1)C_g) \quad (1)$$

$$D_2(u_i, n_j) = \frac{rc}{2} \cdot l_{d,i}^2 + r \cdot l_{d,i} C_g \quad (2)$$

$$D_3(u_i, n_j) = r \cdot (l_{d,i}/2)((1-\gamma+\gamma/2)(c \cdot L(n_j)+(k-2)C_g) \quad (3)$$

$$\text{and } D(u_i, n_j) = D_1(n_j) + D_2(u_i, n_j) + D_3(u_i, n_j) \quad (4)$$

where $\gamma \leq 1$, and note that the $D_1(n_j)$ delay component is the same for all sinks of n_j. The idea behind the 3rd delay component $D_3(u_i, n_j)$ is that without an exact route, we estimate that if u_i lies in the initial γ fraction of the HPBB of n_j starting from the driver position, then, on the average, half of the interconnect length $l_{d,i}$ lies on the main trunk of the estimated route, and it "sees" the entire wire and sink capacitance of the rest of the $(1-\gamma)$ fraction of the net. Furthermore, incremental pieces of this part of the (u_d, u_i) interconnect on the main trunk can also see incremental portions of the γ fraction of the net and load capacitance, which ultimately results in this interconnect seeing a $\gamma/2$ fraction of the total (load + net) capacitance C_{total}.

We define the *critical delay* $D_c(n_j)$ of n_j as:

$$D_c(n_j) = D_1(n_j) + \sum_{u_i \in critical(n_j)} D_2(u_i) + D_3(u_i).$$

The intent here is to include in D_c only the delays of the set $critical(n_j)$ of sinks of n_j lying on near-critical paths. Note that D_c is really a delay-criticality measure of n_j rather than an actual delay of some component of this net. We define the *allocated slack* $S_a(n_j)$ of net n_j as $S(P_{max}(n_j))/(\#$ of nets in path $P_{max}(n_j))$, where $P_{max}(n_j)$ is the maximum-delay path through n_j, and recall that $S(P)$ is the slack of path P.

How much minimization should be performed to reduce a net n_j's interconnect lengths for optimizing the circuit's critical path delay depends not only on the net's D_c value but also on $S(P_{max}(n_j))$—a net with high D_c value but one lying on a path with relatively high slack should have lower delay optimization priority, and similarly for the reverse case. Furthermore, two nets n_i, n_j on different max-delay paths with similar slacks and similar D_c values, should not necessarily be optimized similarly. The important parameter besides D_c for determining optimization priority is the allocated slack S_a of a net. The rationale for this is as follows. Let the max-delay path through n_i (n_j) have 10 (5) nets in them. If the delay optimization priority were the same for all the nets on $P_{max}(n_i)$ and $P_{max}(n_j)$ due to their similar D_c and path slack values, then the delays on their critical interconnects (assuming only one critical interconnect from the driver to a single critical sink on each of the 15 nets) will be made almost equal. This results in $P_{max}(n_i)$ having twice the delay of $P_{max}(n_j)$, and thereby a higher probability of violating the target clock speed. On the other hand, if the delay cost of each net is made $\propto D_c/S_a$, then in our example, since the S_a for the nets in $P_{max}(n_i)$ are half that of those in $P_{max}(n_j)$, the former will have twice the delay optimization priority (i.e., delay cost) than the latter leading to balanced delays for both critical paths $P_{max}(n_i)$ and $P_{max}(n_j)$.

Based on the above arguments we define the *delay cost* $C_D(n_j)$ of n_j as

$$C_D(n_j) = D_c(n_j)/S_a(n_j)^\beta$$

where β is an exponent of the S_a metric that allows magnification (with $\beta > 1$) or shrinking (with $\beta \leq 1$) of differences in optimization priorities of nets on paths with with varying allocated slacks; we use $\beta = 1$ in our experiments.

[1] Our methods apply to routed placements as well. However, since routing consumes a dominant part of the PD phase, it would be beneficial to perform a quick-and-approximate pre-routing estimate of critical path delays using as-accurate-as-possible net route models and performing TD re-placement before proceeding to the actual routing stage. This will hopefully be beneficial for pre-routing corrections thus saving significantly in design times.

[2] For simplicity of exposition, we assume uniform loads for all sink pins, though clearly our net-delay modeling and methods also apply to non-uniform loads.

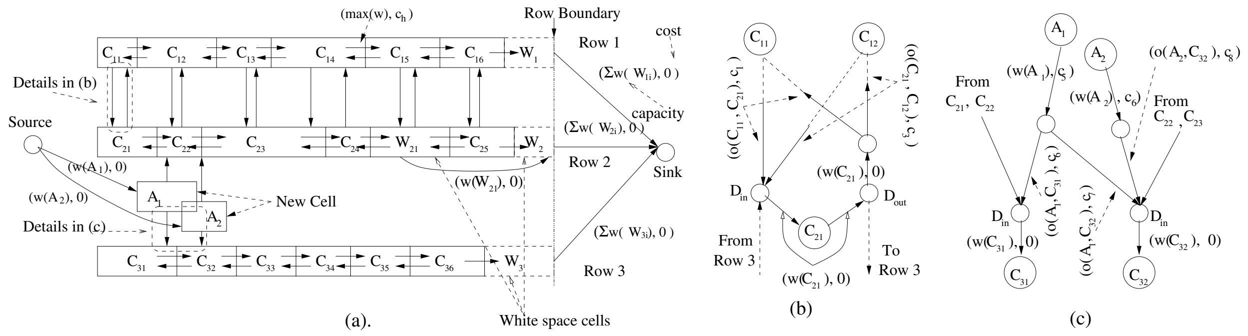

Figure 3: (a) The high-level network flow graph for placing cells A_1, A_2 in legal positions; $w(u)$ is the width of a cell u. (b) Details of flow graph structure for vertical flows between cell pairs $(C_{1,1}, C_{2,1})$ and $(C_{1,2}, C_{2,1})$; $o(u, v)$ is the amount of horizontal overlap between cells u and v. This flow graph structure only allows a flow of amount $<= w(u)$ into a row cell u, and also the vertical flow out of a cell v to go to all cells in the adjacent row that it horizontally overlaps. (c) Similar details of the flow graph structure for flows from the new cells into vertically adjacent row cells.

Note that the $D_c(n_j)$ metric has a component $D_{c,quad}(n_j)$ that is quadratic and a component $D_{c,lin}(n_j)$ that is linear in length metrics. Thus we can write

$$C_D(n_j) = (D_{c,quad}(n_j) + D_{c,lin}(n_j))/S_a(n_j)^{\beta}.$$

The desired TD objective function then is:

$$\sum_{n_j \in moveN} (D_{c,quad}(n_j) + D_{c,lin}(n_j))/S_a(n_j)^{\beta} \quad (5)$$

where recall that $moveN$ is the set of nets connected to cells in $moveC$, the set of cells selected for replacement for reducing delays in critical and near-critical paths.

Since we use a quadratic placer, we need to have a quadratic version of $D_{c,lin}(n_j)$, which we do simply by replacing the linear length metrics (e.g., $L(n_j), l_{d,i}$ in it by their quadratic counterparts (e.g., $L^2(n_j), l_{d,i}^2 = (x_d - x_i)^2 + (y_d - y_i)^2$). Let us call this modified component $D_{c,lin_quad}(n_j)$. Then, the objective function for TAN is:

$$\sum_{n_j \in moveN} (D_{c,quad}(n_j) + D_{c,lin_quad}(n_j))/S_a(n_j)^{\beta} \quad (6)$$

In TAN we optimize the quadratic portion just like in Gordian, and obtain the desired optimization of the linear $D_{c,lin_quad}(n_j)$ as in Gordian-L by dividing $D_{c,lin_quad}(n_j)$ by its current linear value in an inner loop as explained in Sec. 3.1. Note that since we are performing both quadratic and linear optimization, in the inner loop the quadratic-optimization terms remain part of the optimization function without modification (unlike the linear optimization terms). Furthermore, since the analytical placement phase will be followed by a legalizing detailed placer, we do not perform the hierarchical partition-based optimization process of Gordian and Gordian-L.

4. TD NETWORK FLOW BASED DETAILED PLACEMENT

The output of TAN will generally be an illegal placement, but it presents a good starting point for our TD network-flow based detailed placer TIF to place the new cells in legal positions to minimize critical path delays. To accommodate new cell placement, existing cells will be moved minimally. All cell movements are done using TD costs which are: a) proportional to the *delay sensitivities* $D_s(u)$s—$D_s(u)$ is the delay change per unit displacement of u of the most critical interconnect through it, and b) inversely proportional to the *allocated slacks* $S_a(u)$s—$S_a(u) = S_a(n_j)$ where n_j is the net on the max-delay path through u; further details are in Sec. 4.3. Besides placing the new cells in legal positions in a timing-driven manner, TIF also satisfies white space (WS) constraints using novel techniques. The rest of this section describes various aspects of TIF.

4.1 Network Flow Model

Fig. 3(a) shows a generic network flow graph with arc costs and capacities, and a minimum cost flow of some amount x from the source node S to the sink node T that passes through the network. Network flow has found application in VLSI CAD problems ranging from partitioning to placement [4, 5, 16].

Our network flow-based incremental placement algorithm TIF is novel in the way it models arc costs, in that it is timing driven, and in that it accurately solves white space constraints for standard cell placement by overlaying constraints on the flow determination process. The basic network flow model for our detailed incremental placer is shown in Fig. 3(a). Formally, the network graph we use is $F(V, A)$ defined as follows. The node set V is the set $moveC \cup rowC \cup IWS \cup rowWS \cup \{S, T\}$, where $moveC$ is the set of new cells that need to be "pushed" to legal row positions so as to minimize critical path delay, $rowC$ is the set of existing cells in each row of the placement, IWS is the set of intermediate row "WS cells", and $rowWS$ is the set of row WS nodes, one per row, representing the total WS available in each row. The arc set A is given by $pushA \cup vertA \cup horA \cup IWSA \cup rowWSA$, where $pushA$ is the set of flow pushing arcs from S to each cell in $moveC$, $vertA$ and $horA$ are the sets of vertical and horizontal arcs, respectively, that represent cell movements in corresponding directions when flows pass through them, $IWSA$ is the set of arcs going from intermediate WS cells to the corresponding row WS nodes, and $rowWSA$ contains the arcs that go from each row WS node to the sink T. The purpose of these different classes of nodes and arcs in $F(V, A)$ are explained below.

There is a push arc from the source S to each new cell v of capacity the width $w(v)$ of v, and for each such v, there are two vertical arcs from it directed toward cells in rows immediately above and below it (there are more details to these "conceptual" arcs shown in Fig. 3(c)); the capacity of each vertical arc is also $w(v)$. A total flow of $f = \sum_{v \in moveC} w(v)$ emanates from S, and a max-flow solution though the network results in each new cell being pushed to one of its row-position choices (modeled by the vertical arcs from it).

From each row cell, there are two vertical and two horizontal arcs, one in each direction. The vertical arcs from u go to cells in adjacent rows and model possible movement of u in the respective vertical directions; the capacity of these arcs is

$w(u)$, since only u can move along these arcs. The horizontal arcs from u model possible horizontal movement of u within its row, and are potentially of capacity equal to the width of the row from u to the corresponding end of the row, since u could be moved up to either end of the row. However, since arc cost estimates become more inaccurate for large displacements of the cells, a capacity equal to the maximum of the widths of the cell in adjacent rows or new cells that have vertical arcs into u is imposed on its outgoing horizontal arcs. This allows enough horizontal flow through u to cause its movement that remove overlaps with cells vertically moved to its position (via vertical flows into u). There can be intermediate white space within rows and these are modeled as nodes ($\in IWS$) with incoming horizontal and vertical arcs, but each with only one outgoing arc ($\in IWSA$) to the row WS node W_i of the row; the arc's cost is zero and capacity equal to amount of that intermediate white space. Finally, the *total white space* $w(W_i)$ of row i (R_i) = (max row size constraint) - ($\sum$ [cell widths in it]) is also modeled as a node W_i at the right end of the row with an incoming horizontal arc from the rightmost cell and an outgoing arc ($\in rowWSA$) to T of zero cost and capacity = $w(W_i)$.

4.2 The Simplex Network Flow Algorithm

The Simplex method is widely used to solve min-cost max-flow problems. Its basic idea is to iteratively improve an initial solution. It starts with a feasible but generally non-optimal flow of the given amount f. After that, it tries to find *negative cycles*, defined as cycles that have negative costs when traveling in a certain direction. For each such cycle, the Simplex method augments or pushes a flow of the maximum possible value in the cycle in the negative-cost direction. It continues doing so until there are no negative cycles, or flows in negative cycles cannot be further augmented because the capacity of some arc in each cycle is either full in the direction of the flow or there is no flow on some arc in the reverse direction. Our implementation is based on the Simplex algorithm in [3].

4.3 Arc Cost Functions

As mentioned earlier, the TD cost of arc (u, v) should be: i) proportional to the delay change or sensitivity of the most critical interconnect of its start node u to unit length displacements of u in the direction of the arc, and ii) inversely proportional to the allocated slack of its start node u. Delay sensitivity, which is essentially the derivative of the delay function w.r.t. start cell displacement, is a good measure of performance cost when cells are moved by not-very-large displacements from well-established positions, as in the case of incremental detailed placement.

Eqns. 1-4 give the delay formulation for a sink u_i on net n_j. The sensitivity of this delay to a displacement of either sink u_i or driver u_d by $\Delta l_{d,i}$ can be obtained by taking derivatives w.r.t. $l_{d,i}$, and following the components in Eqns. 1-4, these are:

$$\Delta D_1(u_i, n_j) = R_d c \cdot \Delta L(n_j) \approx R_d c \cdot \Delta l_{d,i} \quad (7)$$

$$\Delta D_2(u_i, n_j) = rc \cdot l_{d,i} \cdot \Delta l_{d,i} + r \cdot \Delta l_{d,i} C_g \quad (8)$$

$$\Delta D_3(u_i, n_j) = \Delta D_{3a}(u_i, n_j) + \Delta D_{3b}(u_i, n_j), \text{where}$$

$$\Delta D_{3a}(u_i, n_j) = r \cdot (\Delta l_{d,i}/2)((1-\gamma/2)(c \cdot L(n_j) + (k-2) C_g) \quad (9)$$

$$\Delta D_{3b}(u_i, n_j) = r \cdot (l_{d,i}/2)((1-\gamma/2)(c \cdot \Delta l_{d,i}) \quad (10)$$

$$\Delta D(u_i, n_j) = \Delta D_1(u_i, n_j) + \Delta D_2(u_i, n_j) + \Delta D_3(u_i, n_j). \quad (11)$$

Note that the $\Delta l_{d,i}$ can be positive or negative based on the movement of the cell in question (u_d or u_i) in the direction of the arc e whose cost is being determined. The magnitude of $\Delta l_{d,i}$ for a horizontal arc is its capacity (which reflects the maximum displacement of the cell), and for a vertical arc, it is the spacing between the two adjacent rows that the arc spans (this reflects the exact cell displacement if there is any positive flow along this arc).

The displacement of a cell u in the direction of a flow arc e emanating from it impacts critical nets connected to u in two ways: a) as a sink on the most critical net connected to it, and b) as a driver of the most critical net connected to it.
a) As a sink, there are two cases:
i) u is the most critical sink of its most critical net n_j, in which case its effect on the delay change on n_j is

$$\Delta D_a(u) = \Delta D(u, n_j) \text{ as explained in Eqns. } 7-11.$$

ii) u is not the most critical sink of its most critical net n_j, in which case its effect on the delay change on n_j is

$$\Delta D_a(u) = \Delta D_1(u, n_j) + \Delta D_{3b}(u_i, n_j),$$

which reflects the displacement's effect on $L(n_j)$ and thereby on $\Delta D(u_i, n_j)$ for the most critical sink u_i on n_j.
b) As a driver of its most critical net n_k, the effect of $u's$ displacement on the delay on its most critical interconnect is:

$$\Delta D_b(u) = \Delta D(u, n_k) \text{ given by Eqn. 11}$$

Based on the above, the cost of an arc e (i.e., its unit-flow cost) emanating from u is:

$$cost(e) = (\Delta D_a(u) + \Delta D_b(u)/cap(e)) \cdot \frac{1}{S_a(u)^\kappa}$$

Note that $S_a(u) = S_a(n_j) = S_a(n_k)$ as n_j and n_k lie on the max-delay path through u, and κ is a variable exponent to magnify or shrink cost differences among arcs emanating from cells connected to critical and non-critical nets; $\kappa = 2$ gives us the best overall results.

4.4 Tackling Illegalities in the Incremental Placement DOP

As mentioned earlier, the core incremental detailed placement problem is a DOP, and thus certain illegalities are introduced in it by using a continuous optimization method like network flow. We discuss two main illegality issues and their *in-processing* techniques that we have developed, i.e., techniques that work simultaneously with the network-flow algorithm.

4.4.1 Discrete flow requirement in vertical arcs

Figure 4(b) shows a vertical arc (u, v) from cell u to v of capacity $w(u) = 5$ and unit-flow cost c_1. This arc is used to model the possible movement of u to the row immediately above it (and thus to the position of v). The physical interpretation of any flow along (u, v) has to be that u is moved to v's location, since any position in between its current position and that of v's is illegal. Thus the exact requirement of the flow amount through (u, v) should be either 0 (no movement of u) or $w(u) = 5$. Furthermore, any flow of $x < w(u)$ through (u, v) will also incur an inaccurate lower cost of $x \times c_1$ rather than the "full cost" of $w(u) \times c_1$, incurred in actually moving u to $v's$ position. The resulting inaccuracies in cell movements implied by such flows are shown in Figs. 4(b-c).

We rectify these inaccuracies, by initially having a capacity of 1 and cost = $w(u) \times c_1$ (the full cost) for (u, v) as illustrated in Fig. 4(d). When a flow of 1 passes through (u, v) correctly incurring the full cost of (u, v), we update (u, v)'s capacity to

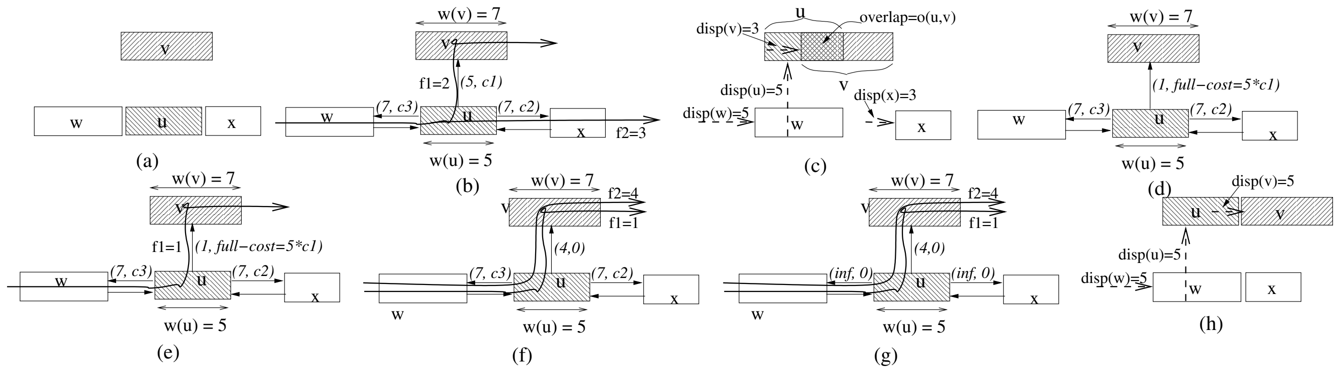

Figure 4: (a) Initial placement; (b) "Regular" cost and capacity of vertical arc (u, v) and two flows through cell u. (c) The physical translation of this flow leading to inaccurate incremental placement of affected cells. (d)-(h) New cost, capacity structure of arc (u, v) with dynamic update, resulting in a flow more closely mimicking the corresponding physical movement of cells, and the final accurate incremental placement of affected cells in (h). The dashed arrows in (c) and (h) represent displacements of cells at the end of the arrows.

$w(u) - 1$ and cost to 0, thus correctly allowing an additional flow of $w(u) - 1$ to pass through it at no cost. Note also that any flow entering u can exit from either the two horizontal arcs or the two vertical arcs including (u, v). Note that even with a flow of 1 through (u, v), in the physical interpretation we will move u to v's position, and thus v will be shifted to it's left or right by a distance of $w(u)$ to remove its overlap with u. Also, the resulting costs of these movements in the incremental placement of the cells affected by the flow of 1 through (u, v) will be incurred, irrespective of whether or not there is any more flow on (u, v). Hence, for the rest of the flow coming into u, if any, we encourage $w(u) - 1$ of it to go through (u, v) at 0 cost by maintaining positive costs for the two horizontal arcs from u as shown in Fig. 4(e-f), as well as for the other vertical arc out of u. Only after a flow of $w(u) - 1$ passes through (u, v), do we make the cost of the horizontal arcs 0 (since u is no longer in this row) and their capacity ∞; see Fig. 4(g). Fig. 4(h) shows the correct cell movements implied by the resulting flow of Fig. 4(g).

As a final point, we note that whenever an arc e's cost and capacity are updated, appropriate updates are made to various entities so that the correct list of negative cycles are available for cost reduction in the current max-flow.

4.4.2 Split flows

Since a flow on a horizontal or vertical arc out of a cell u represents movement of u in the direction of the arc, as far as the incremental placement DOP is concerned, a flow into u can exit from at most one outgoing arc of u.

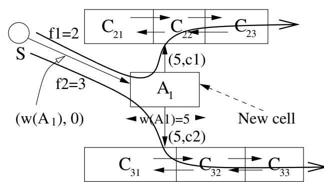

Figure 5: Split flow through new cell $A1$.

Such a requirement, and in general completely legal cell placement can be accurately modeled by an integer quadratic programming (IQP) formulation. However, the IQP problem is well-known to be NP-hard [6], and such a formulation of the incremental placement problem would be intractable. The continuous optimization solution we obtain to this problem via the network flow model is in P and much faster. Of course, it has no restriction on how many outgoing arcs from a node can have a flow, resulting in what we term *split flows* when more than one output arc from a node has positive flows; see Fig. 5.

We have used two alternative heuristics to remedy split flows, and thereby obtain legal cell placements, after an initial min-cost max-flow through the network:

- *Min-cost heuristic*: For each cell u with an outgoing split flow, follow each outgoing flow to the sink or up to a certain distance from u, and determine the minimum-cost of such paths among those from all the outgoing flows. Set capacities of all other outgoing arcs to zero (other than the arc that is contained in the min-cost path). Perform another min-cost max-flow through the new network.
- *Max-flow heuristic*: For each cell u with an outgoing split flow, determine which arc has the maximum flow. Set capacities of all other outgoing arcs to zero. Perform another min-cost max-flow through the new network.

Our experiments revealed that the max-flow heuristic performed much better in terms of the min-cost metric, than the min-cost heuristic for a distance of 1. We thus use the latter heuristic in all our network-flow formulations.

4.5 Satisfying White Space Constraints

It would seem that row white space constraints are automatically satisfied due to the structure of flows through intermediate WS cells and the row WS node which have outgoing arc capacities equal to the amount of WS they represent. However, problems may arise due to the non-binary nature of flows through vertical arcs as discussed in Sec. 4.4. Referring to Figs. 4(e-f), assume that u is in row i R_i, v in row $i-1$ R_{i-1}, and that the total WS in R_{i-1}, $w(W_{i-1})$, is 3. A total flow of $f = 2$ (note that a total flow of 5 depicted in Figs. 4(e-f) may not be available) coming from the left of R_i into u and then to v and then right into the WS node of R_{i-1}, and finally to the sink T will be allowed. However, if that is the only flow on arc (u, v), then the problem comes in the translation of this flow into cell movements[3]—when u is actually moved up to R_{i-1}, since $w(u) = 5$, there will actually be a WS violation in R_{i-1} of $w(u) - w(W_{i-1}) = 2$.

Thus corrective measures are needed subsequent to one min-cost max-flow iteration followed by its physical translation in order to remove violations from rows wherever they exist. On a different, but as we will see related, note, even though there may be enough WS in, say, R_i, to accommodate all new cells that have vertical arcs into R_i, due to the finite capacity of horizontal arcs (e.g., the horizontal arcs from u in Fig. 4(b) have a capacity of 7), not all these cells can be moved into R_i in a single min-cost max-flow iteration. Thus a series of min-cost max-flow iterations are performed to push new cells into nearby rows, as well as to correct WS vio-

[3]Note that the overlap of u and v shown in Fig. 4(c) is not the issue here, as given enough WS in R_{i-1}, v and subsequent cells to its right can be moved to the right to remove all overlaps without violating the WS constraint in R_{i-1}.

lations (from previous iterations) in rows where they exist, while minimizing TD costs.

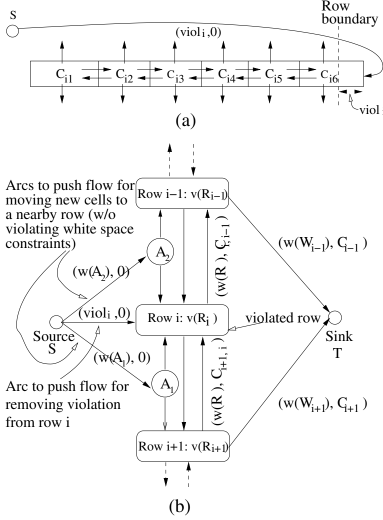

Figure 6: **(a) Part of the detailed flow graph for violation correction in row i. (b) The global flow graph for a combination of row violation correction and insertion of new cells into legal row positions.**

Fig. 6(a) shows an arc going from the source S to the rightmost cell of a row R_i with violation $viol_i$ in order to push out a cell of size at least $viol_i$ from the row. Thus a detailed flow graph will have a structure that is a combination of that shown in Fig. 3(a) and discussed in Sec. 4.1 for non-violating rows, and violation correction arcs from S to violating rows shown in Fig. 6(a). Flow iterations in the detailed flow graph can be very time-intensive due to the large numbers of nodes and arcs in it. However, if we can find a good direction for *global flows* to go between rows and from rows to the sink T (via their WS arcs), then, instead of solving the flow problem at the level of detail of the myriad number of individual arcs, we can refine such a fast global flow by following it with a more precise flow in a subgraph of the detailed flow graph induced by the global flow.

The global flow graph (see Fig. 6(b)) is the directed graph $F_g(V_g, A_g)$ where $V_g = moveC \cup \{v(R_i)|v(R_i)$ is a node representing row $R_i\} \cup \{S, T\}$, where recall that $moveC$ is the set of new or movable cells, and A_g contains: vertical arcs between the row nodes $v(R_i)$'s of adjacent rows and from the new cells to the $v(R_i)$'s of their adjacent rows, push arcs from the source S to the new cells (as in the detailed flow graph), violation-correction arcs from S to WS-violating row nodes, and finally arcs from row nodes with WS to the sink T. The capacities and costs of relevant arcs are shown in Fig. 6(b). The capacity of a vertical arc between two row nodes $v(R_i)$ and $v(R_{i-1})$ is $w(R)$, the maximum row size, and its cost $C_{i,i-1}$ is the weighted average of the detailed vertical arc costs between the two rows. The capacity of an arc from $v(R_i)$ to T is the WS $w(W_i)$ in the row, and its cost C_i is the probabilistic average of all left-to-right detailed horizontal arc costs in R_i [4].

The iterations of alternating global and detailed flows are shown in Fig. 7. Using the combination of global-detailed flow graphs gave us a run-time reduction by about 65% compared to using only a detailed flow graph, at the cost of about 1-2% delay deterioration.

5. EXPERIMENTAL RESULTS

All TD benchmarks we have created and all placement outputs of FlowPlace and Dragon along with a pre-routing STA tool are available at [7]. We used three benchmark suites in our experiments: 1) The TD-Dragon suite of [15], 2) Faraday benchmarks from [2] and 3) TD versions of the IBM benchmark suite that we have constructed from [1]. The first set of benchmarks has complete cell and timing information. The second set has no cell delay information. For the IBM benchmarks, only cell size and net lists are given, so we have to identify FFs. We do this by identifying cycles, and then choose one cell in each cycle to be a FF. To minimize the # of FFs, we choose cells that can break the most # of cycles as FFs. Furthermore, if any path has an excessive length (more than 220), we determine more FFs on such paths to reduce their lengths to lie in the range 180-220. The resultant percentage of FFs is about 13%. Because of the lack of cell delay information in the latter two benchmarks, cell delays are set to zero. Also, because the Faraday and IBM benchmarks include macro cells which we do not handle now, all macro cells are changed to standard cells with a W/H ratio of 4:1. All benchmarks are initially placed by Dragon (the TD Dragon benchmarks are also placed by TD-dragon which can only place this suite)[5]. We then identify paths with delays of at least 90% of the max-path delay as critical paths. The γ value (Eqn. 3) is set to 1. Except for TD-Dragon benchmarks which are run with whitespace (WS) range from 3-10% since these circuits are relatively small, all other benchmarks are given a 3% WS[6]. Table 1 shows various characteristics of the placed benchmarks. For TD-Dragon benchmarks initially placed by Dragon, we collected data both with and without cell delays. Electrical parameters we use are for 0.18 μm: $r = 7.6 \times 10^{-2}$ ohms/μm, $c = 118 \times 10^{-18} f/\mu$m, $R_d = 1440$ ohms, and $C_g = 10^{-15} f$; for TD-Dragon benchmarks, R_d and C_g are derived from their timing library files and are similar to the above values. The unit length for the IBM benchmarks was taken as 0.1125 μm, and that for the Faraday ones as 0.0005 μm[7]. Results were obtained on Linux and Windows XP Pentium IV machines with up to 1GB of main memory, and almost the same program execution speeds.

We first establish the appropriateness of our pre-route net delay estimates of Sec. 3.2. Comparing our estimated delays in Table 1 for TD-Dragon benchmarks with their routed delays given in [15], the four corresponding delay pairs in ns are (5.1, 3.8), (6.2, 4.29), (4.0, 3.39) and (8.2, 6.7), with our values given first. We can see that our delays are generally only 22% larger, and that there is good fidelity between the two delays. Table 2 shows that with the initial placement done by the state-of-the-art WL-driven placer Dragon, FlowPlace achieves up to 34% and an average of 18.3% delay improvement with less than 8% WL deterioration. Table 3 shows that even starting with circuits placed by a TD placer, we can improve results appreciably—with 10% WS, we get up to about 10% and an average of 6.17% delay improvement.

```
while not (all new cells pushed and all rows
free of WS violations) do begin
1.  Construct global flow graph;
2.  Determine a min-cost max-flow in it;
3.  Construct a subgraph of the detailed flow
graph induced by the global flow;
4.  Determine a min-cost max-flow in it;
5.  Perform translation of the detailed flow
into corresponding cell movements;
end while.
```

Figure 7: **Alternating global and detailed flow algorithm.**

[4]The probability of a horizontal arc (x, y) in the detailed flow graph being crossed by a flow that goes right through R_i and directly into T is given by (distance of right boundary of x from left boundary of row)/(row length), which assumes that a flow into the row can come in at any point with uniform probability.

[5]All placement results reported here including that of FlowPlace are without row spacing, as is also the case for results in [14, 15]. FlowPlace's delay improvements with row spacing are a little better (by 2-4.5%) than without row spacing; see [7].

[6]A WS of α % means that the max allowable row size for FlowPlace = $(1 + \alpha/100)$ (max row size of placed input).

[7]These values are chosen so that the cell heights in μm are roughly the same as in the TD-Dragon benchmarks.

Ckt	# cells	# nets	crit. len.	avg. len.	Init. pl runtime (secs)	delay (ns)	HP BB ($10^5 \mu m$)
ibm01	12506	13636	134	128	402	2.5	1.8
ibm02	19342	19325	147	140	864	3.6	4.2
ibm03	22853	27118	113	102	1283	2.3	5.5
ibm04	27220	31679	121	119	1501	3.8	6.7
ibm05	28146	27490	21	45	1594	1.44	10.1
ibm06	32332	34654	113	110	1800	4.7	5.6
ibm07	45639	47786	107	107	2216	2.2	9.3
ibm08	51023	50227	14	49	5973	2.1	9.7
ibm09	53110	60606	125	130	4032	4.3	11.0
ibm10	68685	74179	178	174	4578	4.1	18.1
ibm11	70152	81402	127	128	4415	2.6	16.2
ibm12	70439	76313	181	182	4850	10.3	23.5
ibm13	83709	99106	134	128	5189	3.7	19.6
ibm14	147088	152138	211	201	7432	5.5	38.1
ibm15	161187	186218	202	194	7629	9.1	50.0
ibm16	182980	189259	192	184	7714	12.2	53.1
ibm17	184852	188503	220	203	8259	17.2	79.1
ibm18	210341	201640	71	79	9454	8.0	50.2
TDmatrix	3083	3200	73	65	254	5.1	1.0
TDvp2	8714	8789	75	69	966	6.2	3.4
TDmac32	8902	9115	29	27	1634	4.0	4.0
TDmac64	25616	26017	39	37	6854	8.2	20.4
matrix	3083	3200	60/71	62/64	70	0.5/5.2	0.9
vp2	8714	8789	16/69	20/65	161	2.5/6.7	3.2
mac32	8902	9115	20/25	19/25	184	3.0/4.4	3.9
mac64	25616	26017	36/37	33/36	1364	7.2/9.1	17.8
DMA	11734	11815	16	37	384	1.2	2.1
DSP1	26301	27590	24	58	1527	2.7	4.3
DSP2	26281	27574	25	57	1602	2.7	4.3
RISC1	32622	33186	12	57	1952	3.3	5.9
RISC2	32622	33186	12	59	1906	4.1	6.1

Table 1: Placed benchmark circuit characteristics. "crit. len." is the # of cells in the most critical path, and "avg. len." is the average numbers of cells among all critical paths. The benchmark names with TD as the prefix were placed by TD-Dragon and the rest by Dragon. The "delay" column gives the critical path delay of the placed benchmarks. For TD-Dragon benchmarks placed by Dragon, two values are given for some attributes: the left one is the data ignoring cell delay, while the right one considers cell delay. Over all benchmarks, the avg. runtime for Dragon is 3270 secs, while the average runtime for TD-Dragon for the TD-Dragon circuits is 2427 secs.

We also used the linear-delay model of [14, 15] to compare more directly to their techniques. Results for this model are under the "TDD model" column—we obtain, with 5% WS, up to 7.9% and an avg. of 3.4% improv. over TD-Dragon, while [14], with 10% WS, obtained an avg. of 2.8% improv. over TD-Dragon. FlowPlace is also quite fast: it completes incremental placement in about 17.5% of the placement time of Dragon and in about 12% of the time of TD-Dragon. WL deterioration on the avg. over all benchmark suites is $< 8\%$.

6. CONCLUSIONS

We have presented various novel and effective techniques for TD incremental placement. These include: a) pre-routed net-delay estimates with good fidelity; b) delay-sensitivity and allocated-slack based flow arc costs; c) simultaneous quadratic and linear optimization in our TD analytical placer; d) novel global and detailed flow graph structures for performing TD cell placement and white-space constraint satisfaction; and e) in-processing techniques for correction of illegalities arising from solving a discrete optimization problem (TD incremental placement) by a fast continuous optimization method. The end-result is a robust, effective and efficient TD incremental placer FlowPlace that achieves significant delay improvements in quick time on circuits placed by state-of-the-art WL (Dragon) and TD (TD-Dragon) placers. FlowPlace also scales well with circuit size; e.g., we obtained about 34% delay improvement for a 210K cell circuit ibm18 in about 24 minutes with a WL deterioration of only 2.6%.

7. REFERENCES

[1] S. N. Adya, I. L. Markov, "Consistent Placement of Macro-Blocks using Floorplanning and Standard-Cell Placement", *Proc. ACM/IEEE Intl. Symp. on Physical Design*, 12-17, April, 2002

[2] S. N. Adya, et al., "Unification of Partitioning, Floorplanning and Placement",*ICCAD*, 2004, pp. 41-46.

[3] R. Ahuja, et al., "A network simplex algorithm with O(n) consecutive degenerate pivots", *Operations research letters*, pp. 1417-1436, 1995.

Ckt	#of mv cells	-Δ%WL aft. TAN	+Δ%delay aft. TAN	final -Δ%WL	final +Δ%delay	runtime (secs)
ibm01	311	2.12	19.4	9.9	15.1	203
ibm02	589	2.34	34.0	10.1	26.3	201
ibm03	512	2.30	33.1	9.7	30.3	338
ibm04	689	1.98	26.1	9.3	21.5	484
ibm05	551	0.12	21.6	4.3	18.0	304
ibm06	529	2.33	24.9	8.9	20.4	402
ibm07	968	0.52	18.9	6.7	14.1	729
ibm08	684	-0.74	30.7	3.8	28.4	795
ibm09	1312	3.12	12.1	9.7	8.4	854
ibm10	989	1.48	19.3	8.7	14.9	811
ibm11	953	2.85	20.7	10.2	14.1	621
ibm12	1338	-0.14	19.4	5.1	13.8	724
ibm13	928	2.50	20.9	8.3	18.4	916
ibm14	1241	1.88	18.3	7.3	13.2	979
ibm15	1197	0.12	21.8	4.7	18.8	1142
ibm16	1428	-0.15	25.1	4.3	20.5	1198
ibm17	1897	0.10	27.6	2.6	25.6	1304
ibm18	2168	0.10	36.9	2.6	34.1	1451
Avg.	**985**	**1.09**	**24.2**	**7.1**	**19.7**	**747**
DMA	354	3.01	24.9	12.8	19.1	297
DSP1	324	2.17	25.1	10.8	21.3	294
DSP2	379	2.11	24.1	10.2	20.8	302
RISC1	601	1.97	24.6	9.7	22.2	331
RISC2	584	2.33	22.8	10.5	19.8	387
Avg.	**448**	**2.32**	**24.3**	**10.9**	**20.6**	**322**
matrix	277	1.39	9.98	9.68	7.12	231
vp2	359	1.02	10.57	4.51	5.01	284
mac32	332	0.99	14.25	8.24	10.54	381
mac64	406	1.67	13.24	8.11	10.04	376
Avg.	**343**	**1.26**	**12.01**	**7.63**	**8.17**	**318**
nmatrix	254	0.82	17.24	12.91	11.2	377
nvp2	248	1.33	34.18	12.05	30.4	594
nmac32	325	1.54	22.06	3.25	18.9	161
nmac64	389	2.21	22.97	4.84	18.0	248
Avg.	**304**	**1.47**	**24.11**	**8.20**	**19.62**	**345**
Overall Avg.	**727**	**1.35**	**22.62**	**7.92**	**18.34**	**571**

Table 2: FlowPlace results for circuits initially placed by Dragon. The last 4 results are for TD-Dragon benchmarks (placed by Dragon) ignoring cell delays. -Δ%WL is the WL deterioration % & +Δ%delay is the delay improv. %. The "# of mv cells" is $|moveC|$. The WS for all results is 3%.

Ckt (TD)	#of mv cells	3%WS +Δ%del	5%WS +Δ%del	10%WS +Δ%del	TDD model +Δ%del	avg. -Δ%WL	runtime (secs)
matrix	256	1.18	3.07	4.01	0.01	8.07	208
vp2	372	0.69	0.92	3.02	2.32	9.75	269
mac32	251	4.75	5.12	7.70	3.31	2.89	319
mac64	364	7.51	7.79	9.98	7.91	2.77	328
Avg.	**310**	**3.53**	**4.22**	**6.17**	**3.39**	**5.87**	**281**

Table 3: FlowPlace results for circuits initially placed by TD-Dragon. The last runtime column is the average run time for runs with different % WS's. The TDD model column shows results using the delay model in [15] for 5% WS.

[4] U. Brenner, et al., "Almost optimum placement legalization by minimum cost flow and dynamic programming", *ISPD*, 2004.

[5] K. Doll, F. Johannes, K. Antreich, "Iterative placement improvement by network flow methods", *IEEE Trans. Computer-Aided Design*, pp.1189-1200, 1994.

[6] C. A. Floudas and V. Visweswaran, "Quadratic Optimization", in *Handbook of global optimization*, Kluwer Acad. Publ., Dordrecht, pp. 217–269, 1995.

[7] The FlowPlace page: http://www.ece.uic.edu/~dutt/benchmarks-etc/FlowPlace/flow.html.

[8] A. Kahng, S. Mantik and I. Markov, "Min-Max placement for large scale timing optimization", *ISPD'02*.

[9] A Kahng, I. Markov, S. Reda "Boosting: min-cut placement with improved signal delay" *Proc. DATE*, 2004, pp. 1098-1103

[10] J. Kleinhans, et al., "GORDIAN: VLSI placement by quadratic programming and slicing optimization". *IEEE Trans. CAD*, vol. 10, pp.356-365, Mar. 1991.

[11] A. Marquardt, V. Betz and J. Rose, "Timing-Driven Placement for FPGAs", *Proc. Int'l Symp. FPGAs*, 2000, pp. 203-213.

[12] S-L Ou, M. Pedram, "Timing-driven Placement Based on Partitioning with Dynamic Cut-net Control", *Proc. Design Automation Conf. 2000,* pp. 472-476

[13] G. Sigl, et al., "Analytical placement: A linear or a quadratic objective function?", *Proc. DAC*, 1991, pp. 427-432.

[14] C. Wonjoon, K. Bazargan, "Incremental placement for timing optimization",*ICCAD*, pp. 463-466, 2003

[15] Y. Xiaojian, C. Bo-Kyung, M. Sarrafzadeh; "Timing-driven placement using design hierarchy guided constraint generation", *ICCAD*, pp. 177-180, 2002.

[16] H. Yang and D. F. Wong, "Efficient Network Flow Based Min-cut Balanced Partitioning", *IEEE Trans. CAD*, pp. 1533-1540, 1996.