DREAMPlace 4.0: Timing-driven Placement with Momentum-based Net Weighting and Lagrangian-based Refinement

Peiyu Liao, Dawei Guo, Zizheng Guo, Siting Liu, Yibo Lin, Bei Yu

***Abstract*—Optimizing timing is critical to the design closure of integrated circuits (IC). However, most existing algorithms for circuit placement focus on the optimization of wirelength instead of timing metrics. This paper presents a timing-driven placement framework. It consists of a global placement stage based on net weighting with momentum, and a detailed placement stage based on Lagrangian multipliers. By improving the preconditioners and timing engines to facilitate net weighting and discrete local search, we have achieved superior timing improvement on benchmarks from ICCAD 2015 contest, including worst negative slack (WNS) and total negative slack (TNS).**

I. Introduction

CIRCUIT placement is an important VLSI design stage. Placement aims at finding the optimal locations of circuit components on a given chip layout [2]. Placement is often formulated as a mathematical optimization problem with objective functions minimizing the cost of interconnects between circuit components. In most previous placement frameworks, the interconnect cost is modeled by the total wirelength of all nets, which is estimated by half-perimeter wirelength (HPWL) or other approximations. Besides being only an approximation, total wirelength pays equal attention to all nets instead of focusing on timing-critical nets and paths. This is contrast to timing-driven placement that specifically targets wires on timing-critical paths which often yields immediate circuit performance benefits.

Placement can be divided into a global placement stage and a detailed placement stage, and timing optimization can be applied to both stages. The goal of timing-driven global placement is to achieve both roughly good *worst negative slack* (WNS) and *total negative slack* (TNS). Later, timing-driven detailed placement pays more attention to WNS optimization by perturbing the current placement solution locally around critical paths. There are two types of timing-driven placement optimization: net-based and path-based approaches.

In **net-based** approaches, optimization are done on nets within the design. These approaches translate timing analysis feedbacks into changes of net weights and other constraints in order to optimize critical circuit regions. The weights of nets can be computed statically once before placement optimization based on either slack [3]–[6] or sensitivity [7]–[9] statistics. The drawback of such approaches is that the timing analysis at earlier placement iterations are unreliable due to frequently-changing cell locations, leading to less effective and representative net weights. Such drawback is remedied by updating net weights dynamically across all placement iterations [3], [10]–[12]. In addition to net weighting, timing analysis results can also be used to limit the maximum net lengths, which are called net constraint-based approaches [13]–[17]. The formulation of net constraints varies from particular placers [2].

Contrast to net-based approaches, **path-based** approaches focus on direct optimization of critical timing paths [18]–[21]. They move cells on selected critical paths to explicitly reduce the delay of these paths. Such path-based objective is often formulated as a mathematical programming problem to optimize, and can usually outperform net-based approaches in terms of solution quality. However, as the number of paths can grow near exponentially with the growth of design size, such path-based approaches are poor in their runtime scalability.

Both the net-based and the path-based approaches have strengths and weaknesses regarding different targets such as solution quality and turnaround time, which involve inevitable trade-off during any timing optimization. Generally, we prefer less constrained approaches with knowledge of different placement iterations and more runtime scalability to large designs, especially considering the flexible cell placement formulation during global placement.

In this paper, we propose a timing-driven placement engine with momentum-based net weighting in global placement and Lagrangian-based refinement in detailed placement. The net-based approach is applied because of its scalability to global perturbation of cells in global placement. After the timing-driven global placement, we further integrate a timing-driven detailed placement procedure to improve the timing quality. We summarize the major contributions as follows.

- **Momentum-based net weighting scheme**. The net weighting scheme is crucial in our timing-driven global placement algorithm. At each timing iteration, every net should be assigned a positive weight by incorporating the current slack evaluation within the existing net criticality. The net weights will be gradually updated to keep the timing profile up-to-date by considering the new weights, computed according to slacks, to be a momentum term,

The preliminary version has been presented at the IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE) in 2022 [1]. This work is supported in part by AI Chip Center for Emerging Smart Systems (ACCESS) and The Research Grants Council of Hong Kong SAR (Project No. CUHK14209420). (*Corresponding authors: Yibo Lin and Bei Yu*)

P. Liao, S. Liu and B. Yu are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR.

D. Guo, Z. Guo and Y. Lin are with the School of Integrated Circuits, Peking University, China.

which is analogous to the *momentum method* that is widely used in backpropagation algorithm for neural network training [22].
- **Preconditioning for net weighting**. The preconditioner proposed by the original ePlace [23] algorithm only considers trivial net weights such that every net is equally treated. Providing that critical nets should have higher net weights for timing optimization, the numerical stability may get negatively affected, especially for those cells incident to critical paths. The preconditioner will be enhanced and generalized to adapt non-trivial net weights in placement optimization.
- **Lagrangian-based Refinement**. After the timing-driven global placement, a timing-driven detailed placement inspired from [24] will be performed immediately to further improve the timing quality. We use a high-quality black-box timer to evaluate and analyze timing and iteratively update cell locations.
- Experimental results on the ICCAD2015 contest benchmark suites [25] show that we can achieve about 50% improvements on TNS, and 30% improvements on WNS on average, compared to the state-of-the-art placer [26] after global placement and legalization. After, we can further make roughly 10% improvements on both TNS and WNS in detailed placement. The timing quality comparison is listed in TABLE II.

The rest of the paper is structured as follows. Section II provides some preliminaries including brief foundations of nonlinear placement, static timing analysis, and timing optimization. Section III presents the descriptions of our timing-driven global placement algorithms and the detailed analysis. Section IV presents the timing-driven detailed placement algorithms. After the algorithm sections, Section V demonstrates the experimental results and some related analysis, followed by Section VI summarizing the whole paper.

II. Preliminaries

A. Nonlinear Global Placement

In global placement, a circuit is modeled as a directed graph where nodes represent cells and edges represent interconnects between cells. A typical circuit graph contains millions of nodes to be placed on the chip layout. We let $\mathbb{N} = (V, E)$ denote this graph where V denotes the set of cells and E denotes the set of interconnects, i.e., nets. Placement engines try to assign two location vectors $(\boldsymbol{x}, \boldsymbol{y})$ in $\mathbb{R}^n$ corresponding to all n cells to minimize the total wirelength of nets, denoted as $W(\boldsymbol{x}, \boldsymbol{y})$. With net weights applied, the weighted sum of wirelengths can be formulated as

$$\min_{x,y} \sum_{e \in E} w_e W(e; \boldsymbol{x}, \boldsymbol{y}), \tag{1}$$

where w_e denotes the weight of net $e \in E$. To ease the optimization of wirelength, modern placement engines adopt smoothed approximations to the half-perimeter wirelength (HPWL) model $W(e; \boldsymbol{x}, \boldsymbol{y})$. Smoothed approximations provide gradient information to cell movements which is especially useful in nonlinear placement frameworks which rely on the differentiability of objective functions. One commonly-used smoothed approximation of HPWL is called weighted-average (WA) equations [27], [28] as follows,

$$\begin{aligned} \widetilde{W}_x(e; \gamma; \boldsymbol{x}, \boldsymbol{y}) &= \frac{\sum_{i \in e} x_i e^{\frac{x_i}{\gamma}}}{\sum_{i \in e} e^{\frac{x_i}{\gamma}}} - \frac{\sum_{i \in e} x_i e^{-\frac{x_i}{\gamma}}}{\sum_{i \in e} e^{-\frac{x_i}{\gamma}}}, \\ \widetilde{W}(e; \gamma; \boldsymbol{x}, \boldsymbol{y}) &= \widetilde{W}_x(e; \gamma; \boldsymbol{x}, \boldsymbol{y}) + \widetilde{W}_y(e; \gamma; \boldsymbol{x}, \boldsymbol{y}), \end{aligned} \tag{2}$$

In Equation (2), $\widetilde{W}_x(e; \gamma; \boldsymbol{x}, \boldsymbol{y})$, $\widetilde{W}_y(e; \gamma; \boldsymbol{x}, \boldsymbol{y})$ denote net wirelengths across horizontal and vertical directions, respectively, and the hyperparameter γ controls the approximation precision. This equation is a drop-in replacement for the wirelength model used throughout the rest of this paper. Finally, the non-linear placement objective is formulated as

$$\min_{x,y} \left(\sum_{e \in E} w_e W(e; \boldsymbol{x}, \boldsymbol{y}) + \lambda D(\boldsymbol{x}, \boldsymbol{y}) \right), \tag{3}$$

where $D(\cdot, \cdot)$ denotes the density penalty to encourage cell spreading and λ denotes the penalty multiplier.

To fully leverage the advantages of gradient-based numerical optimization methods, the objective in Equation (3) should be everywhere differentiable and its gradient should be feasible to compute numerically. The wirelength model $W(e; \boldsymbol{x}, \boldsymbol{y})$ is a nonlinear approximation, so optimizing Equation (3) is known as nonlinear global placement.

B. Static Timing Analysis

The basis of a successful timing-driven placement lies in fast and accurate timing computation through static timing analysis (STA). STA evaluates the delay-annotated circuit timing graph under worst-case and best-case scenarios, and then compute its setup and hold timing performance. During this process, the signal arrival time and constraints are propagated on the graph through logic paths [29].

Specifically, the delay-annotated timing graph is directed and acyclic, with each node denoting a circuit pin and each edge denoting a directed connection between circuit pins where signal can pass through. The signal propagation involves both a forward propagation and a backward propagation process, respectively computing arrival times and required arrival times of pins. For any pin p with arrival time denoted as $t_{\text{at}}(p)$ and required arrival time denoted as $t_{\text{rat}}(p)$, we can define its *slack* as the difference between them,

$$s(p) = t_{\text{rat}}(p) - t_{\text{at}}(p). \tag{4}$$

Slack is a critical and widely-used indicator of timing quality around every pin evaluated. The timing performance of a whole placement solution is an ensemble of slacks on endpoint pins. By picking the smallest negative slack values, we get the most commonly used *worst negative slack* (WNS),

$$s_{\text{wns}} = \min_{t \in P_{\text{end}}} s(t), \tag{5}$$

here P_{end} denotes the set of endpoint pins including flip-flop (FF) inputs and output ports, and s_{wns} denotes the WNS value. We assume at least 1 pin with negative slack value,

i.e., $\exists t \in P_{\text{end}}$ s.t. $s(t) < 0$, which means there exists timing violations in current placement iteration. Apart from WNS, *total negative slack* (TNS) [17] is another well-known timing objective which sums all negative endpoint slacks instead,

$$S_{\text{tns}} = \sum_{t \in P_{\text{end}}} s(t)^{-} = \sum_{t \in P_{\text{end}}, s(t)<0} s(t). \tag{6}$$

C. Timing Optimization

Timing-driven placement aims at improving the TNS and WNS of the underlying design. While both metrics depict the timing quality, they pay attention to different aspects of the circuit timing behavior. WNS cares about the single most critical signal path by definition, whereas TNS involves the critical paths at all timing endpoints, possibly taking tens of thousands of paths into consideration that expands to the whole circuit topology. As a result, TNS incorporates a more global view of timing optimization opportunities which turns out to be useful in timing-driven global placement, whereas WNS is more emphasized during detailed placement.

A complete timing-driven placement formulation with both objectives and constraints can be presented as following,

$$\begin{aligned} \max \quad & s(\boldsymbol{x}, \boldsymbol{y}) \\ \text{s.t.} \quad & \rho_b(\boldsymbol{x}, \boldsymbol{y}) \le \rho_t, \forall b \in B. \end{aligned} \tag{7}$$

In Equation (7), the objective $s(\boldsymbol{x}, \boldsymbol{y})$ is a slack function that we need to maximize, i.e., minimizing the absolute value of slacks. It can be either WNS or TNS, or the combination of them. The constraints are related to cell density which is sampled from the $m \times m$ planar grids (i.e., bins) denoted as B on the circuit layout. In each bin $b \in B$, $\rho_b(\boldsymbol{x}, \boldsymbol{y})$ and ρ_t denote the current density and the target density, respectively. To encourage cell spreading, we try to stop density overflows. The timing-oriented objective $s(\boldsymbol{x}, \boldsymbol{y})$ replaces the total wirelength objective used in previous wirelength-driven analytical placement algorithms while leaving cell constraints untouched. Contrary to the wirelength functions with closed-form analytical representations, slack metrics hardly incorporate explicit forms. This leads to our choice of an indirect timing optimization scheme by net weighting.

III. Timing-Driven Global Placement

The overall flow of the proposed placement framework including both global placement and detailed placement is illustrated in Fig. 1 in detail. Tasks of placement placement are described in red boxes, while tasks of detailed placement are in cyan boxes. In this section, we focus on the timing-driven global placement.

At the global placement stage, we expect to integrate static timing analysis into the iterations of cell location updates during the gradient-based optimization so that the placement solution can be optimized in terms of timing. More specifically, we expect the timing analysis tool to give us feedback about how good the current placement is, which paths have an intensive impact in terms of timing, and how such information can affect the cell locations during the placement. We have to determine whether the timing analysis should be performed at each optimization step compared to modern gradient-based analytical placers. If the current iteration is regular, we skip the timing optimization block and check convergence criteria directly. Otherwise, the current iteration is determined to be a *timing iteration*, which launches the net weighting process.

The timing optimization in global placement consists of three consecutive steps: *RC tree construction*, *static timing analysis* and *net weighting*.

A. RC Tree Construction

Constructing RC trees is a crucial step required by our timer to perform timing analysis. At the placement stage, we only have cell locations without routing information, so we must provide a policy to leverage coordinates and model timing propagation. More specifically, for each net, we are supposed to construct an RC tree that roots from its driving pin and connects each sink pin.

Additionally, RC trees should be re-constructed for all nets at every timing iteration, as the cell locations will change in every backward step. Since it is computationally expensive to construct RC trees at every gradient-based iteration in placement optimization, two adjacent timing iterations should not be too close. Therefore, the pin distribution of the same net may vary significantly in two timing iterations, which forces us to re-construct RC trees for reliability.

Given a *possibly illegal* placement solution, the pin locations of all nets are provided. Each time we perform the timing analysis, the pin locations are considered fixed, until the next cell location update iteration.

For each net, we start with the pins it connects. A FLUTE [30] call will be performed to construct the *rectilinear Steiner minimal tree* of this net. The Steiner tree generally reflects the internal timing propagation inside the timer we use. With the help of Steiner trees, we can roughly model the routing solution by inserting Steiner points. For any driving-sink pair in a net, there exists a unique path in the corresponding Steiner tree, which provides hints to calculate inter-connect net delays.

Considering that the placement solutions will only affect the inter-connect timing modeling, we can apply Elmore's delay model [31] which is sufficient for timing-driven placement. We illustrate the RC tree construction process from a simple 4-pin net in Fig. 2(a) and Fig. 2(b). The generated Steiner tree is visualized in Fig. 2(a). In contrast, the abstract RC tree hierarchy is described in Fig. 2(b).

In Fig. 2(a), we have a simple 4-pin net that connects pins p_1, p_2, p_3 and p_4, where p_1 is the driving pin. The FLUTE [30] will give us a Steiner tree rooting from p_1, connecting sinks with two Steiner points s_1 and s_2 added. The Steiner nodes are inner nodes of the RC tree in Fig. 2(b), while a pin is either a root or a leaf.

To build RC information from inter-connections, we require the resistance value per unit length r' and the capacitance value per unit length c' pre-determined for the given design. Each line segment with length l contributes a resistance value $r'l$ and a capacitance value $c'l$.

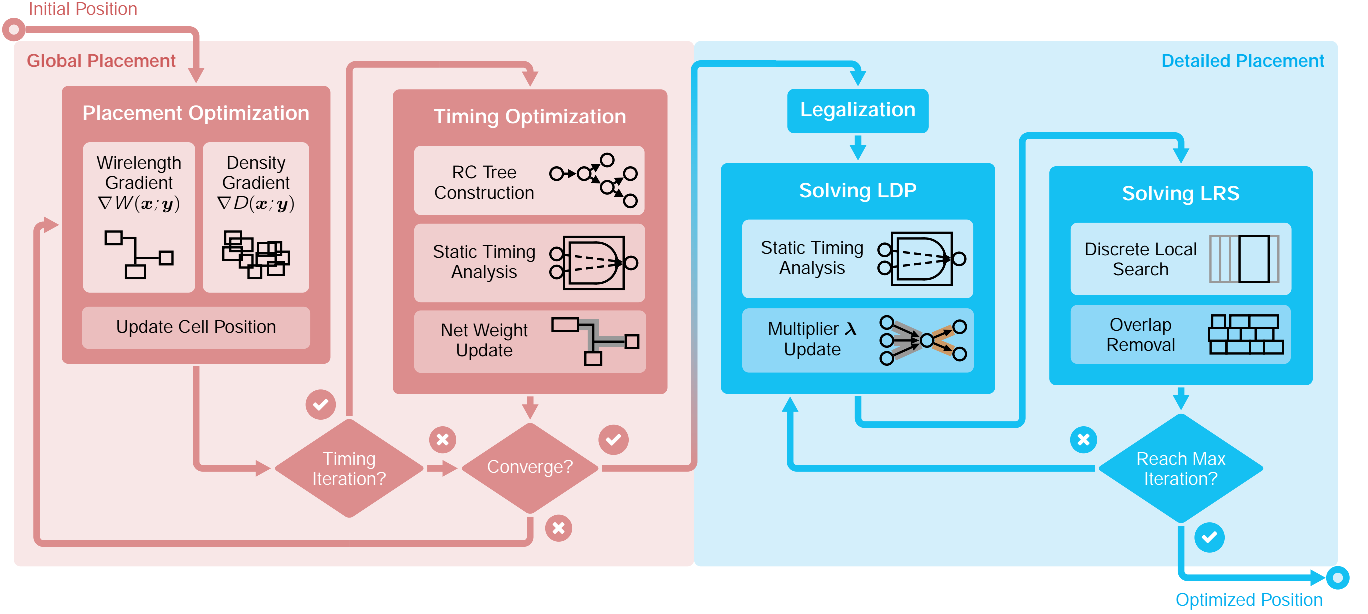
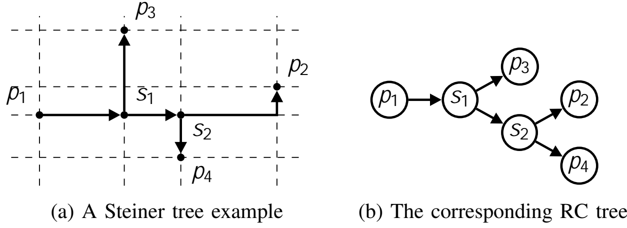


Fig. 1 The overall flow of placement including both global placement and detailed placement with timing optimization. Tasks of placement placement are described in red boxes, while tasks of detailed placement are in cyan boxes.



(a) A Steiner tree example (b) The corresponding RC tree

Fig. 2 A 4-pin net example of a Steiner tree and its corresponding RC tree constructed for net delay calculation.

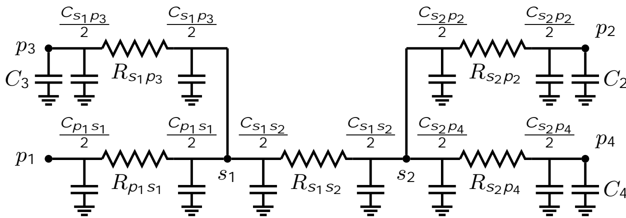


Fig. 3 The Elmore delay model for the above 4-pin net example.

B. Static Timing Analysis

The details can be enriched in Fig. 2(b) by adding several abstract resistors and capacitors for edge segments in the RC tree, illustrated in Fig. 3. In our framework, Elmore's delay model [31] is applied to approximate actual delays. More specifically, we use model to break wires into RC sections. After we fill the RC information into the RC tree initialized by the timer, we then naturally proceed to the static timing analysis.

The resistance and capacitance values can be directly computed according to the Manhattan distances given pin coordinates. Note that cell overlaps exist in global placement, so the coarse-grained inter-connect timing model is enough to catch the general criticality information.

The static timing analysis is crucial to obtain the timing profile at any timing point. It is performed at every timing iteration so that we can keep the timing profile up-to-date. Once the slack values are correctly calculated, we then smoothly proceed to the net weighting part, which updates the net weights of all nets.

C. Momentum-based Net Weighting

Every net is assigned a weight in the objective function of wirelength-driven placement. Some prior knowledge of net wirelength contribution can be a hint fed into the wirelength-driven placer by adjusting net weights. A net with a higher weight will be more sensitive to the updates of cell locations, as a perturbation to its total wirelength will lead to a greater impact on the objective we are optimizing. The optimizer will implicitly tend to place cells such that the bounding boxes of nets with higher weights can be smaller.

Critical nets have a more pronounced impact on the final timing performance. Reducing the HPWL of a critical net will lead to a more significant gain in timing. Without any doubt, critical nets should be reasonably assigned higher weights to guide the placer to place cells incident to them closer.

Net Criticality. We define *criticality* value in our placement database as a guide to update net weights. The higher criticality value of a net, the more critical it will be in timing analysis. Therefore, we should assign higher criticality values to those critical nets according the timing analysis report at each timing iteration. For a specific net e, let c_e and $\mathcal{S}_{\text{wns}}$ denote its criticality value and the worst negative slack (WNS) of the circuit design, respectively. The WNS $\mathcal{S}_{\text{wns}}$

directly comes from the timing report. On the other hand, the criticality value $c_e^{(m)}$ of net e at the m-th timing iteration is calculated iteratively based on the historical value $c_e^{(m-1)}$ and the *momentum* of the current timing iteration. The momentum of criticality value of a net e is defined as

$$c_{mom,e} = \begin{cases} 0, & \text{if } s_{wns} \geq 0; \\ \left(\dfrac{s_e}{s_{wns}}\right)^+, & \text{otherwise}, \end{cases} \tag{8}$$

where s_e is the net slack of e. In Equation (8), function $x^+ = \max\{x, 0\}$ is the *positive part* of any real number $x \in \mathbb{R}$. It is also known as the rectified linear unit (ReLU) activation function in neural network training.

If the WNS s_{wns} is non-negative, the net weights will all remain unchanged. Otherwise, s_{wns} is negative, and the criticality value of net e is defined as the positive part of the slack ratio $\frac{s_e}{s_{wns}}$, which means that only nets with negative slacks will be considered. If a net e has a negative slack $s_e < 0$, its criticality momentum will be set to the ratio $\frac{s_e}{s_{wns}} = \frac{|s_e|}{|s_{wns}|}$. For net e, The higher the absolute slack value $|s_e|$ we compute, the higher criticality momentum $c_{mom,e}$ it will have according to Equation (8).

Intuitively, the net criticality should indicate the probability of net e to be critical. Since the timer will report different timing-critical paths at each iteration, we should also update the criticality values iteratively. A critical net may be related to multiple critical paths, and different nets may have different negative slacks. Hence, the criticality update policy should be strongly correlated to the negative slacks. Nets with higher absolute slack values are thought to be more sensitive to timing metrics, so we are supposed to assign higher net weights to them accordingly so that the placer will try to shrink its bounding box.

Consider a specific net e. Define its criticality value at the m-th iteration as $c_e^{(m)}$. From Equation (8), we know that its net slack $s_e^{(m)}$ and the WNS $s_{wns}^{(m)}$ can be calculated at each timing iteration by static timing analysis. Then we obtain its momentum criticality value $c_{mom,e}^{(m)}$ of net e at the m-th iteration, which corresponds to the criticality updates.

Net weighting Scheme. We introduce a momentum-based net weighting scheme. Consider a specific net e in the design. The net weight of e is always positive (a net with zero weight is ignored), so we decide to take its logarithm to discuss. Let $\tilde{w}_e^{(m)} = \log w_e^{(m)}$ and $\Delta\tilde{w}_e^{(m)}$ be the logarithmic net weight of w_e and its increment at the m-th timing iteration, respectively.

$$\tilde{w}_e^{(m+1)} = \tilde{w}_e^{(m)} + \Delta\tilde{w}_e^{(m)}. \tag{9}$$

Note that the base of the logarithm can be customized. In fact, the addition operation in Equation (9) is equivalent to a multiplication without the logarithm. This way, we can guarantee the positiveness of the net weights.

Considering that the criticality momentum at different timing iterations are independent of each other, we expect the net weight $w_e^{(m)}$ to be emphasized by its criticality $c_e^{(m)}$, which accumulates the current criticality momentum. For any integer

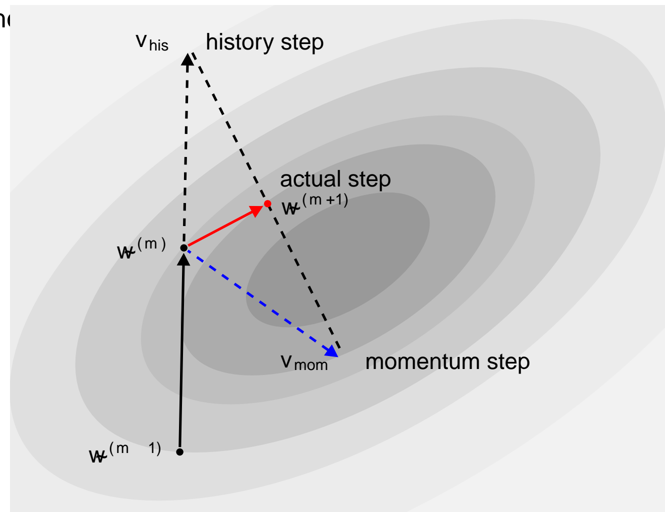


Fig. 4 The visualization of a simple example illustrating the mechanism of how the momentum vectors will affect the actual step.

$m \geq 0$, the increment and the logarithmic criticality can be modeled by

$$\begin{aligned} \Delta\tilde{w}_e^{(m)} &= \tilde{c}_e^{(m)}; \\ \Delta\tilde{w}_e^{(m+1)} &= \Delta\tilde{w}_e^{(m)} + \tilde{c}_{mom,e}^{(m)}; \end{aligned} \tag{10}$$

where the logarithmic criticality and its momentum are defined as

$$\begin{aligned} \tilde{c}_e^{(m)} &= \log(1 + c_e^{(m)}); \\ \tilde{c}_{mom,e}^{(m)} &= \log(1 + c_{mom,e}^{(m)}). \end{aligned} \tag{11}$$

The decay coefficient is a hyperparameter within $[0, 1]$. > 0 is also a positive hyperparameter indicating the contribution of the momentum term in the new weight increment. The term $\Delta\tilde{w}_e^{(m)}$ can be considered as the velocity, from Equation (9).

The scheme in Equations (9) and (10) is inspired by the momentum-based gradient descent algorithm on backpropagation during neural network training. In a typical backpropagation of training when optimizing $f(w)$ where w represents the weights, the momentum term should be ∇f so that the actual gradient increment Δw value can be guided while remembering the history update at each iteration. More specifically, one may use $\Delta w^{(m)} - \nabla f$ as the actual increment in the next iteration, where and are interpreted as the *decay factor* and the *step size*, respectively.

In net weighting, the net criticality may be unstable during global placement. Therefore, here we apply the momentum step to the update of the criticality values and the net weights to smooth the increment of logarithmic net weights. If a net e has a positive criticality $c_e > 0$, its weight should be increased according to the criticality magnitude. Besides, if a net is reported to be critical at most timing iterations, it may have a large net weight as w_e will keep increasing at most timing iterations. The weight differences are acceptable as long as no value overflow is reported.

We would like to adopt the notations in matrix calculus where we use boldface to represent vectors, then the scheme described in Equations (9) and (10) can be reformulated as

$$\Delta \mathbf{w}^{(m+1)} = \beta \Delta \mathbf{w}^{(m)} + \gamma \mathbf{e}^{(m)}_{mom}; \qquad (12)$$

where $\mathbf{w}^{(m)}$, $\Delta \mathbf{w}^{(m)}$, and $\mathbf{e}^{(m)}_{mom}$ indicate the logarithmic net weights, their increments, and the transformed momentum vector calculated by Equation (8), respectively, at the m-th timing iteration. The update rule Equation (12) is similar to the aforementioned formula $\beta \Delta \mathbf{w}^{(m)} - \eta \nabla f$. All vectors in Equation (12) have the same size that is exactly the total number of nets in the design. A simple example illustrating the mechanism of momentum-based net weighting works is shown in Fig. 4.

If the criticality momentum $e^{(m)}_{mom,e}$ of net e has a very small magnitude in the late period of global placement, the net weight increment $\Delta w^{(m)}_e$ will approximately decay by the factor β with the increment of iteration m, and consequently the net weight $w^{(m)}_e$ will stabilize gradually. Therefore, we can keep highlighting nets remaining critical during global placement according to Equation (12). Here, the parameter γ is simply determined by $1 - \beta$ to make Equation (12) a linear combination.

Different from [10] and other similar dynamic net weighting schemes, all nets are considered instead of those only on critical paths. We do not analyze and extract the critical paths explicitly in global placement. Instead, the nets with more negative slack values are thought to be critical as they will be assigned higher criticality values according to Equation (8).

D. Preconditioning

Preconditioning is very critical to numerical optimization. A conventional preconditioner usually aims at solving the inverse matrix of the Hessian matrix $\mathbf{H}_f^{-1}$ for unconstrained optimization $\min f(\mathbf{x})$, so as to implement the exact Newton direction $-\mathbf{H}_f^{-1}\nabla f$. In real applications, the Hessian matrix is never computed explicitly due to the huge computational overhead. Therefore, Hessian-free methods are preferred in most of numerical optimization techniques.

In global placement problems, the industrial designs are very likely to contain millions of instances, so it is impossible to calculate the exact Hessian matrix at each global placement iteration. The ePlace [23] preconditioner simply ignores non-diagonal entries $\frac{\partial^2 f}{\partial x_i \partial x_j}$, which is actually nonzero in real applications. Eliminating all non-diagonal entries makes the Hessian transformation $\mathbf{H}_f^{-1}\nabla f$ an element-wise scaling on the gradient ∇f.

The objective function f is set to Equation (3) by default. Without loss of generality, we only consider the horizontal cell locations $\mathbf{x} \in \mathbb{R}^n$ here. The i-th diagonal entry of the Hessian matrix $\mathbf{H}_f^{-1}$ is given by

$$\frac{\partial^2 f}{\partial x_i^2} = \sum_{e \in E} w_e \frac{\partial^2 W(e; \mathbf{x}; \mathbf{y})}{\partial x_i^2} + \lambda \frac{\partial^2 D(\mathbf{x}; \mathbf{y})}{\partial x_i^2}; \qquad (13)$$

where w_e is the net weight of net e, and D is the density penalty of the circuit given the current placement.

In the ePlace [23] algorithm, the second-order derivative term $\frac{\partial^2}{\partial x_i^2} W(e; \mathbf{x}; \mathbf{y})$ is computationally expensive due to the complicated form of the weighted-average model [27]. We also adopt this approximation. More specifically, the second derivatives will be binary and only when node $v_i \in V$ is incident to net $e \in E$ will the term be set to 1. This rule of thumb will approximate the wirelength term in Equation (13) as

$$\sum_{e \in E} w_e \frac{\partial^2 W(e; \mathbf{x}; \mathbf{y})}{\partial x_i^2} \approx \sum_{e \in E_i} w_e; \qquad (14)$$

where E_i is the net subset incident to $v_i \in V$. Note that the net weighting scheme only affect the inter-connects and the wirelength term, so we are allowed to follow the same computational approximation as [23] for preconditioning.

$$\frac{\partial^2 D(\mathbf{x}; \mathbf{y})}{\partial x_i^2} = q_i \frac{\partial^2 \psi_i(\mathbf{x}; \mathbf{y})}{\partial x_i^2} \approx q_i; \qquad (15)$$

where q_i is the quantity of electrical charge of the node $v_i \in V$. This coarse-grained approximation can save huge computational overhead. The approximate preconditioning matrix on the horizontal direction will be

$$\tilde{\mathbf{H}}_{f_{x;x}} = \mathrm{diag}\left(\sum_{e \in E_1} w_e + \lambda q_1; \cdots; \sum_{e \in E_n} w_e + \lambda q_n \right). \qquad (16)$$

If the net weights are trivial, i.e., every net has a weight of 1, $\sum_{e \in E_i} w_e$ will be degraded to $|E_i|$ which represents the total number of nets incident to v_i. Together with the vertical direction, the preconditioned gradient vector will be $\nabla f_{precond} = \tilde{\mathbf{H}}_f^{-1} \nabla f$.

IV. Timing-Driven Detailed Placement

Global placement provides a roughly good solution from scratch. Compared to global placement with a clear form of numerical optimization, detailed placement refines standard cell locations locally to further improve specific objectives. The normal wirelength-driven detailed placement adopts multiple discrete methods to further improve circuit wirelength. In the timing-driven detailed placement, the timing metrics become the major objective. In this section, we further improve the negative slacks via iterative local search.

We use $N = (V; E)$ to represent the set of net list where V is the set of standard cells. Let V_{PI} and V_{PO} be the set of primary inputs and primary outputs. For any standard cell (node) $v_i \in V$, we use $a^L_i = t^L_{at}(p_i)$ and $a^E_i = t^E_{at}(p_i)$ to denote the late and early arrival times of the output p_i of v_i. Similarly, we use $r^L_i = t^L_{rat}(p_i)$ and $r^E_i = t^E_{rat}(p_i)$ to denote the late and early required arrival times of the output p_i of v_i. In addition, let d^L_{ij} and d^E_{ij} denote the late and early delay values from node v_i's output to node v_j's output, where v_i : $v_i \in F_j$, i.e., node v_i is a fan-in of $v_j \in V$. In timing-driven detailed placement, timing objectives should be considered more directly and explicitly, so that we have more accurate estimation and are able to perform precise refinement.

The general timing-driven detailed placement can be formulated as follows.

$$
\begin{aligned}
\min \quad & -\sum_{j:v_j\in V_{PO}} \hat{s}_j^E - \sum_{j:v_j\in V_{PO}} \hat{s}_j^L \\
\text{s.t.:} \quad & \hat{s}_j^E \le 0; \qquad \hat{s}_j^L \le 0; \qquad \forall j: v_j \in V_{PO} \\
& a_j^E - r_j^E \ge \hat{s}_j^E; \quad a_j^L - r_j^L \le -\hat{s}_j^L; \quad \forall j: v_j \in V_{PO} \\
& a_i^E + d_{ij}^E \ge a_j^E; \quad a_i^L + d_{ij}^L \le a_j^L; \quad \forall (i,j): v_i \in F_j \\
& \text{displacement \& legality constraints}
\end{aligned}
\tag{17}
$$

Here notation $\hat{s} = \min\{s, 0\}$ simply means negative slacks. Note that F_j should be empty for $v_j \in V_{PI}$, so the primary inputs are actually not included in the above constraints. The formulation in Equation (17) from [24] aims at optimizing the TNS objective, which is more intuitive than WNS. The latter one is much more sensitive, and thus much harder to optimize. However, a success of improving TNS will usually also lead to the benefit of WNS.

The formulation in Equation (17) is complicated as we have introduced a lot of auxiliary variables $\hat{s}, a, r$ defined for valid nodes $v_j \in V$. There are various applications in gate sizing facing such a kind of optimization problem with complicated constraints [32], [33]. A most widely adopted strategy is to relax the timing constraints into objectives and solve it via Lagrangian dual. More specifically, each constraint in Equation (17) is assigned a Lagrangian multiplier to formulate the objective $L(x, y, s, a, r, \lambda)$ (simplified as L in the following sections),

$$
\begin{aligned}
L = & -\sum_{j:v_j\in V_{PO}} (1-\hat{\lambda}_j^E)\hat{s}_j^E - \sum_{j:v_j\in V_{PO}} (1-\hat{\lambda}_j^L)\hat{s}_j^L \\
& + \sum_{j:v_j\in V_{PO}} \lambda_j^E(\hat{s}_j^E - a_j^E + r_j^E) + \sum_{j:v_j\in V_{PO}} \lambda_j^L(\hat{s}_j^L - r_j^L + a_j^L) \\
& + \sum_{j:v_j\in V}\sum_{i:v_i\in F_j} \lambda_{ij}^E(a_j^E - a_i^E - d_{ij}^E) + \lambda_{ij}^L(a_i^L - a_j^L + d_{ij}^L) .
\end{aligned}
\tag{18}
$$

The typical Lagrangian dual is to solve $\max_{\lambda \ge 0} \min_{x,y} L$. However, the complicated formulation prevents us from solving it feasibly. Note that the auxiliary variables appear in multiple terms. We rearrange the summation order and get

$$
\begin{aligned}
L = & \sum_{j:v_j\in V_{PO}} (\lambda_j^E + \hat{\lambda}_j^E - 1)\hat{s}_j^E + \sum_{j:v_j\in V_{PO}} (\lambda_j^L + \hat{\lambda}_j^L - 1)\hat{s}_j^L \\
& + \sum_{j:v_j\in V} \left(f_j^L(\lambda)a_j^L - f_j^E(\lambda)a_j^E\right) + \sum_{j:v_j\in V_{PO}} \left(\lambda_j^E r_j^E - \lambda_j^L r_j^L\right) \\
& + \sum_{j:v_j\in V}\sum_{i:v_i\in F_j} \left(\lambda_{ij}^L d_{ij}^L - \lambda_{ij}^E d_{ij}^E\right);
\end{aligned}
\tag{19}
$$

where the helper function $f_j^L(\lambda)$ for index j such that $v_j \in V$ is defined as

$$
f_j^L(\lambda) = \begin{cases} \lambda_j^L - \sum_{i:v_i\in F_j} \lambda_{ij}^L; & \text{if } v_j \in V_{PO}; \\ \sum_{k:j\in F_k} \lambda_{jk}^L - \sum_{i:v_i\in F_j} \lambda_{ij}^L; & \text{otherwise}, \end{cases}
\tag{20}
$$

and $f_j^E(\lambda)$ is defined similarly.

For any auxiliary variables $\hat{s}_j^E; \hat{s}_j^L; a_j^E; a_j^L; r_j^E; r_j^L$, where $v_j \in V$, the corresponding term in Equation (19) is linear. The delay values d_{ij}^E and d_{ij}^L are considered to be completely determined by a placement solution (x, y).

Equation (19) is thought to be continuous but indifferentiable. Consider the Karush-Kuhn-Tucker (KKT) optimality conditions of Equation (19). It is not difficult to obtain the complementary slackness condition $\lambda_j^E + \hat{\lambda}_j^E = \lambda_j^L + \hat{\lambda}_j^L = 1$ for any $v_j \in V$ when taking partial derivatives with respect to the auxiliary variables $\hat{s}_j^E; \hat{s}_j^L$ for every valid j. Additionally, we have the flow conservation [34] when taking partial derivatives with respect to $a_j^E; a_j^L; r_j^E; r_j^L$ for every valid j:

$$
\begin{aligned}
\sum_{i:v_i\in F_j} \lambda_{ij}^E &= \sum_{k:j\in F_k} \lambda_{jk}^E; \quad \forall j: v_j \in V\setminus(V_{PI}\cup V_{PO}); \\
\sum_{i:v_i\in F_j} \lambda_{ij}^L &= \sum_{k:j\in F_k} \lambda_{jk}^L; \quad \forall j: v_j \in V\setminus(V_{PI}\cup V_{PO}); \\
\lambda_j^L &= \sum_{i:v_i\in F_j} \lambda_{ij}^L; \quad \forall j: v_j \in V_{PO}; \\
\lambda_j^E &= \sum_{i:v_i\in F_j} \lambda_{ij}^E; \quad \forall j: v_j \in V_{PO}.
\end{aligned}
\tag{21}
$$

Any local minimum of Equation (17) must satisfy the above flow conservation in Equation (21). Now that any local optimum $(x^*, y^*, s^*, a^*, r^*, \lambda^*)$ must admit a multiplier variable λ^* satisfying Equation (21), we directly reduce the solution space in subproblem $\min_{x,y} L$ by combining Equation (19) and Equation (21) as the aforementioned auxiliary variables and their corresponding multipliers can be reasonably cancelled out.

$$
\begin{aligned}
L(x, y, \lambda) = & \sum_{j:v_j\in V_{PO}} \left(\lambda_j^E r_j^E - \lambda_j^L r_j^L\right) \\
& + \sum_{j:v_j\in V}\sum_{i:v_i\in F_j} \left(\lambda_{ij}^L d_{ij}^L - \lambda_{ij}^E d_{ij}^E\right) .
\end{aligned}
\tag{22}
$$

Equation (18) and Equation (22) are not generally equivalent. However, they will be equivalent given a fixed λ satisfying the flow conservation in Equation (21). Compared to the original one, there are much less variables in Equation (22) as the auxiliary variables are cancelled out. Hence, the new Lagrangian dual program should be solving $\max_{\lambda\in\Omega} \min_{x,y} L(x, y, \lambda)$ where set Ω is defined as $\Omega = \{\lambda \ge 0 : \lambda \text{ satisfies Equation (21)}\}$.

Note that the first term $\sum_{j:v_j\in V_{PO}} \lambda_j^E r_j^E - \lambda_j^L r_j^L$ in Equation (22) will not be affected by the movable cells, as the summation is taken over weighted required arrival time values on primary outputs which should be determined by the timing constraints. In other words, we can naturally ignore the first term and focus on the second term. Therefore, we must have

$$
\begin{aligned}
L(\lambda) = & \sum_{j:v_j\in V_{PO}} \left(\lambda_j^E r_j^E - \lambda_j^L r_j^L\right) \\
& + \min_{x,y} \sum_{j:v_j\in V}\sum_{i:v_i\in F_j} \left(\lambda_{ij}^L d_{ij}^L - \lambda_{ij}^E d_{ij}^E\right) .
\end{aligned}
\tag{23}
$$

Now, the target is to solve $\max_{\lambda\in\Omega} L(\lambda)$. Typically, a numerical method should solve $L(\lambda)$ and $\max_{\lambda\in\Omega} L(\lambda)$ separately. The former problem is called the Lagrangian

relaxation subproblem (LRS) and the latter one is called the Lagrangian dual problem (LDP) in [24]. Unfortunately, we are unable to numerically find the optimal or suboptimal solutions of LRS or LDP like what we have shown in global placement. The general heuristic flow includes:

i) a discrete local search to approximately solve LRS;
ii) an overlap removal step to remove cell overlap;
iii) an overall multiplier update according to STA results.

We will introduce the three steps in detail one by one in the following subsections.

A. Discrete Local Search

Unlike the timing model applied in [24], we generally obtain the delay values in Equation (23) from the STA tool black box. Any timing analysis engine can be integrated into the local search to solve LRS problem.

Local Cost. From Equation (23), the discrete local search tries to minimize the second term of Equation (22) as a subproblem

$$\sum_{j:v_j\in V}\sum_{i:v_i\in F_j}\left(\lambda^L_{ij}d^L_{ij}-\lambda^E_{ij}d^E_{ij}\right) = \sum_{(i,j):v_i\in F_j}\left(\lambda^L_{ij}d^L_{ij}-\lambda^E_{ij}d^E_{ij}\right) = \frac{1}{2}\sum_{j:v_j\in V} c_j(x,y;\lambda), \quad (24)$$

where the summation is taken over all pairs (i,j) that $v_i\in F_j$, and the cost $c_j(x,y;\lambda)$ (simplified as c_j) of v_j is defined as

$$c_j = \sum_{i:v_i\in F_j}\left(\lambda^L_{ij}d^L_{ij}-\lambda^E_{ij}d^E_{ij}\right) + \sum_{k:j\in F_k}\left(\lambda^L_{jk}d^L_{jk}-\lambda^E_{jk}d^E_{jk}\right). \quad (25)$$

In Equation (25) we consider both the fanins and fanouts of v_j. Therefore, each valid index pair (i,j) will be counted twice, and that is why we need a $\frac{1}{2}$ factor in Equation (24). However, generally minimize it from a global perspective is intractable. A feasible workaround is to separate the global optimization $\min_{x,y}\sum_{j:v_j\in V} c_j$ into mini-problems $\min_{x_j,y_j} c_j$ for every movable node v_j. In this way, we move the movable nodes one by one and solve the subproblem discretely from a local perspective.

In the local search step, we fix the multipliers. When moving a node v_j, the delay values $d^L_{ij}, d^E_{ij}, d^L_{jk}, d^E_{jk}$ including interconnect delays and the cell delays for valid indices i,k will change accordingly, leading to the update of cost c_j. Since the relationship between cell location (x_j,y_j) and c_j is indirect, we enumerate all locations inside a window surrounding the current location (x_j,y_j) to find the best one (x_j,y_j) with the minimum cost c_j. Fig. 5 shows a simple example of the local search algorithm.

Delay Estimation. In each iteration of our detailed placement algorithm, the location of one node v_j is updated, and the circuit timing parameters such as $d^L_{ij}, d^E_{ij}, d^L_{jk}$, and d^E_{jk} are updated correspondingly. As there are a lot of iterations, the timing update must be done efficiently. The timing update consists of two tasks: net parasitics update and timing propagation update. Both tasks incur a long runtime in current STA engines, which pose difficulty on fast and agile detailed placement flow.

To solve the aforementioned problems, we propose a fast timing update flow to assist our detailed placement iterations. The flow includes two techniques: *local net estimation* and *incremental timing update*. The key observation is that the movement of one node v_j only affects the nets that are directly connected to it. For example, in Fig. 5, the upstream nets $e_1,e_2\in E$ and the downstream net $e_3\in E$ will be affected when moving $v_j\in V$. Thus, we can approximate the timing impacts by only updating the timing in a local region around the moved node.

The movement of v_j affects the parasitics of the surrounding nets, leading to updates on Elmore delay parameters such as impulse values, load capacitances, and delays. As the Steiner tree generation is expensive, we estimate the change to these values instead of rebuilding the whole routing solution. For an affected net e, we denote its driver pin by p_e at coordinate $v(p_e;x_j,y_j)$. A movement of v_j from (x_j,y_j) to (x_j^0,y_j^0) can change the coordinate of p_e. For every driver or sink pin p in net e locating at $v(p;x_j,y_j)$, we define its scale ratio $r(p;x_j^0,y_j^0)$ as

$$r(p;x_j^0,y_j^0)=\begin{cases}\dfrac{W(e;x_j^0,y_j^0)}{W(e;x_j,y_j)}, & \text{if } p \text{ is } p_e;\\[2ex] \dfrac{\|v(p;x_j^0,y_j^0)-v(p_e;x_j^0,y_j^0)\|_1}{\|v(p;x_j,y_j)-v(p_e;x_j,y_j)\|_1}, & \text{otherwise}\end{cases} \quad (26)$$

In Equation (26), $W(\cdot)$ is the wirelength of e. More specifically, the ratio defined above indicates the scale factor of the driving net wirelength if p is a driver, and otherwise the scale factor of the Manhattan distance between sink p and its driver p_e after the above cell movement.

After directly scaling the load capacity, impulse, and the net delay by the factor defined in Equation (26), we have estimated the net parasitics affected by the local cell movement. We then perform an incremental 2-hop forward timing propagation on all affected nets to obtain the estimated values of all $d^L_{ij}, d^E_{ij}, d^L_{jk}$, and d^E_{jk}. In this way, we avoid the time-consuming full routing and timing update. The proposed estimation is effective as long as the cell movement is small, which is the case in our detailed placement flow.

Local Search Algorithm. The order of moving nodes may heavily affect the placement solution, so a better strategy is to search by the reverse topological order [24]. Note that the delay values d^E_{jk} and d^L_{jk} contain the arc delay of cell v_j itself. All delay values are provided by the timer after a complete static timing analysis. The detailed algorithm of the modified discrete local search algorithm is described in Algorithm 1.

As shown in Algorithm 1, we will find the best location of v_j in the reverse topological order of every selected critical path. Consider the sub-procedure of moving v_j. Assume that there are two fanins and two fanouts shown in Fig. 5. We will retrieve delay values $d_{i_1j}, d_{i_2j}, d_{jk_1}$ and d_{jk_2} from the timer to calculate the cost of the current location (x_j,y_j) according to Equation (25). Note that the arc delay of v_j will be included in d_{i_1j} and d_{i_2j}. We will construct a local search window which is the cyan box in Fig. 5. For example, the dashed lines visualize the top left and the bottom right candidate locations of v_j in the 7×3 search window. The location within the

Algorithm 1 LOCAL SEARCH ALGORITHM

Require: The Lagrangian multipliers ^{E}; ^{L} for early and late timing constraints both, the total number of most critical paths k.
1: Use timer to perform static timing analysis.
2: Extract top k critical paths p_1; ; p_k, where p_i stands for the i-th most critical path;
3: Set the path index $i = 1$;
4: **while** i k **do**
5: **for** movable v_j 2 p_i in the reverse topological order **do**
6: Initialize the best cost c_j $+1$;
7: Initialize the best cell location $(x_j ; y_j)$ to be the current cell location;
8: **for** each candidate location $(x_j ; y_j)$ **do**
9: Estimate delay values d_{ij} for any i such that v_i 2 F_j and d_{jk} for any k such that j 2 F_k;
10: Calculate cost c_j by Equation (25) according to the delay values obtained above;
11: **if** $c_j < c_j$ **then**
12: Update c_j c_j and $(x_j ; y_j)$ $(x_j ; y_j)$;
13: **end if**
14: **end for**
15: Place node v_j to the best cell location $(x_j ; y_j)$;
16: **end for**
17: **end while**
18: **return** the refined cell locations $(x ; y)$;

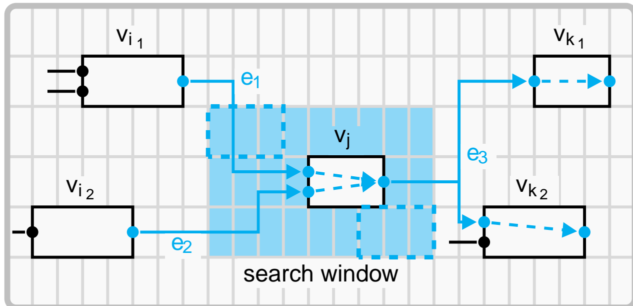


Fig. 5 A simple example illustrating the discrete local search algorithm. The delay values that should be considered when moving cell v_j are marked in cyan. All candidate locations of v_j are included in the search window (the cyan box).

search window with the lowest cost will be selected as the best location $(x_j ; y_j)$.

B. Overlap Removal

Since cell overlap may still exist after the local search, we should perform overlap removal to eliminate illegality. This procedure is a mini-legalization, as legality constraints except cell overlap must be guaranteed.

Inspired by [24], we introduce a fast algorithm to remove cell overlaps while preserving the timing quality given by the discrete local search solution.

The main idea is to iteratively search for the nearest legal location for each cell, reducing the displacement from their previously found locations. In each iteration, we add the candidate locations in the four directions to the search space and find the closest one by managing a min-priority queue sorted by Manhattan distance from their original location.

It was noted that in some circumstances, nodes will inevitably have large displacement when the region around them is dense. In these cases, we prioritize those nodes that be more critical to the timing information to be unaffected. Therefore, unlike [24] which sorted nodes by their center horizontal coordinates, we sort the nodes according to their slacks. Those with large negative slack values are considered as being critical to timing improvement and we will have them fixed first.

Algorithm 2 OVERLAP REMOVAL ALGORITHM

Require: The slacks of movable nodes, the initialized row structures R.
1: Sort movable nodes according to slack values;
2: **for** fixed macro M **do**
3: Insert M in the row structure;
4: **end for**
5: **for** movable v_j 2 V **do**
6: Initialize the min-priority-queue q with current node location $(x_j ; y_j)$;
7: **while** q is not empty **do** $(x_j ; y_j)$
8: **if** $(x_j ; y_j)$ causes overlap **then**
9: Insert locations above, below, left and right of $(x_j ; y_j)$ in q;
10: **else**
11: Place node v_j to the location $(x_j ; y_j)$;
12: Insert $(x_j ; y_j)$ in the row structure;
13: Clear q;
14: **end if**
15: **end while**
16: **end for**
17: **return** the overlap-free solution;

Algorithm 2 shows the detailed procedure for removing overlaps. To reduce the time required by the algorithm, we construct a segment tree, a kind of binary tree storing information of intervals for each row to query or insert the overlap rapidly.

C. Multiplier Update

The values of multipliers are extremely important in the discrete local search process. Therefore, an efficient update strategy can lead to effective timing improvement. The multiplier update solves the LDP subproblem max $_2$ L ( ). Unfortunately, this optimization problem is intractable to solve as we have both the dual feasibility 0 and the flow conservation Equation (21) as constraints. A possible workaround should be like the conventional projected gradient descent. In other words, we make a gradient-descent-like move on , then project the updated multipliers onto the constraint set so that they keep non-negative and satisfy the flow conservation in Equation (21).

TABLE I The statistics of the ICCAD2015 contest benchmarks [25].

Benchmark	#Cells	#Nets	#Pins	#Rows
superblue1	1209716	1215710	3767494	1829
superblue3	1213253	1224979	3905321	1840
superblue4	795645	802513	2497940	1840
superblue5	1086888	1100825	3246878	2528
superblue7	1931639	1933945	6372094	3163
superblue10	1876103	1898119	5560506	3437
superblue16	981559	999902	3013268	1788
superblue18	768068	771542	2559143	1788

[24] uses a modified subgradient method to update the multipliers according to STA results. Multipliers at primary outputs and standard cells are updated according to Equation (27). Based on that, we additionally introduce rise and fall edge information to obtain a more accurate delay estimation since we use a more detailed STA tool.

$$\begin{aligned}
\lambda^{L\,(m+1)}_{ij} &= (a^L_i + d^L_{ij})(a^L_j)^{-1}\lambda^{L\,(m)}_{ij}; && \forall(i,j): v_i \in F_j;\\
\lambda^{E\,(m+1)}_{ij} &= a^E_j (a^E_i + d^E_{ij})^{-1}\lambda^{E\,(m)}_{ij}; && \forall(i,j): v_i \in F_j;\\
\lambda^{L\,(m+1)}_{j} &= a^L_j (r^L_j)^{-1}\lambda^{L\,(m)}_{j}; && \forall j: v_j \in V_{PO};\\
\lambda^{E\,(m+1)}_{j} &= r^E_j (a^E_j)^{-1}\lambda^{E\,(m)}_{j}; && \forall j: v_j \in V_{PO}.
\end{aligned} \tag{27}$$

All the arrival times are extracted from static timing analysis. The timer also calculates the delay values. With the help of Equation (27), the multipliers in the m-th iteration $\lambda^{(m)}$ will be updated to $\lambda^{(m+1)}$ that will be used in the following local search iteration.

After being updated, all multipliers need to be scaled in the reverse topological order according to Equation (21) to satisfy the flow conservation condition [34]. Multipliers on critical paths tend to increase rapidly under the effect of flow conservation, which is in line with our expectation of increasing the net weights on critical path:

$$\hat{\lambda}_{ij} = \frac{\sum_{k:v_j \in F_k} \lambda_{jk}}{\sum_{i:v_i \in F_j} \lambda_{ij}} \lambda_{ij}. \tag{28}$$

The multiplier update in Equation (28) should be executed in the reverse topological order. After a full step of multiplier update, the parameter should satisfy the flow conservation requirement.

V. Experimental Results

A. Experimental Setup

We conducted experiments on the ICCAD 2015 contest benchmark suites [25]. Parameters of the design are shown in TABLE I. All the cases are relative large since most of them contain a great number of cells and nets. Movable macros are not included in any benchmark.

We implemented the proposed algorithm in C++ based on the open-source placement tool DREAMPlace [26] and the open-source timer OpenTimer [35]. Notably, we manage to fully utilized GPU resources in both core placement [26] and timing analysis [36]. For fairness of comparison, we use the same default hyperparameter settings of DREAMPlace [26]

B. TNS and WNS Improvement

It is important to determine when we should set up observation, perform timing analysis and update net weights. Performing a timing analysis in each iteration is not possible, as it introduces a huge overhead. Initially, all cells are concentrated to the center of the layout and their locations are highly overlapped, so that reliable timing analysis results cannot be given until cells are approximately uniform out by density forces. In our experiments, we evaluate timing metrics and update net weights every 15 iterations after the 500th iteration of the global placement. In addition, we update the net weights using the manually customized hyperparameters in Equation (12). We use = 1 and the decay factor is set to 0.5 by default for all benchmarks. Since the large-scale benchmarks may be affected by various factors, the optimal may also vary for different cases.

However, our detailed placement algorithm is highly sensitive to timing metric, since the movement of cells on critical path has a significant impact on the delay and affects the next round of iterations. Therefore, we choose to build RC tree right after discrete local search and overlap removal at each iteration. Additionally, because of the flow conservation condition, multipliers on critical paths may grow exponentially and lead to inaccurate location prediction after too many iterations, so we terminate our algorithm when no slack improvement is observed and take the optimal solution. In our experiments, the optimal solution will appears within 10 iterations in most cases.

Using the evaluation script provided by the ICCAD 2015 contest, we evaluate our global placement and detailed placement solution after legalization. The results are listed in TABLE II. As indicated in the table, compared with the DREAMPlace [26] without any timing-aware optimization, our algorithm can significantly improve both TNS (50.59% on average) and WNS (38.15% on average) after the timing-driven global placement and detailed placement, which confirms the effectiveness of our algorithm.

In addition, we implement the classic dynamic net weighting scheme in [10] for a comparison. This net weighting scheme is used for timing-driven quadratic placement and can The results are listed in the second column of TABLE II. The best result among all results is highlighted using **boldface**, and the second one is colored with brown. As indicated in the table, our algorithm can outperform [10] a lot on TNS, which brings us a positive enlightenment that it is definitely useful considering timing-aware optimization at both global placement and detailed placement stage.

Notably, we also compare our proposed algorithm to the outstanding open-source end-to-end silicon compiler OpenROAD [37] (code available in [38]) in TABLE II. The timing-driven flag is turned on to incorporate net weighting for timing optimization in global placement. OpenROAD [37] has integrated RePlAce [39] as its global placer. The corresponding results of OpenROAD [37] in TABLE II are collected after legalization. In our experiments, we found that the default parameter settings would induce divergence on the ICCAD 2015 contest benchmarks [25], so we slightly decreased the

TABLE II Comparison among DREAMPlace [26], DREAMPlace [26]+ [10], OpenROAD [37], our timing-driven global placement only [1] and our algorithm including both global and detailed placement. The best results are emphasized with boldface, and the second-best results are colored in brown. The TNS unit and the WNS unit are 10^5ps and 10^3ps, respectively.

Benchmark	DREAMPlace [26]		DREAMPlace [26]+ [10]		OpenROAD [37]		Our GP [1]		Our GP + Our DP	
	TNS	WNS	TNS	WNS	TNS	WNS	TNS	WNS	TNS	WNS
superblue1	-252.359	-18.5414	-121.963	-13.1548	-192.369	-28.0059	-85.0315	-14.1031	-67.0417	-13.0626
superblue3	-88.4701	-33.2509	-61.2222	-15.6518	-70.2105	-23.3123	-54.7427	-16.4341	-52.2784	-16.2643
superblue4	-196.498	-21.4654	-177.800	-11.8600	-190.705	-21.6523	-144.380	-12.7808	-136.174	-11.0606
superblue5	-208.943	-48.4825	-108.019	-47.7110	-140.441	-35.7177	-95.7820	-26.7602	-94.0022	-23.7035
superblue7	-161.989	-20.3957	-84.3107	-19.9126	-194.620	-19.0151	-63.8629	-15.2163	-58.3746	-15.2163
superblue10	-839.134	-33.7599	-786.359	-29.0470	-575.778	-23.3536	-768.748	-31.8796	-705.795	-25.3786
superblue16	-438.267	-16.8146	-175.543	-18.5297	-523.530	-18.2700	-124.181	-12.1115	-118.697	-11.2812
superblue18	-90.4280	-20.1261	-69.4700	-11.7831	-75.8221	-9.95419	-47.2458	-11.8705	-43.4910	-11.6939
Average Ratio	2.357	1.667	1.385	1.270	2.176	1.460	1.089	1.090	1.000	1.000

TABLE III The HPWL and Runtime comparison of timing-driven placement among DREAMPlace [26], DREAMPlace [26]+ [10], OpenROAD [37], our timing-driven global placement only [1], and our algorithm including both global and detailed placement. The unit of Runtime is second.

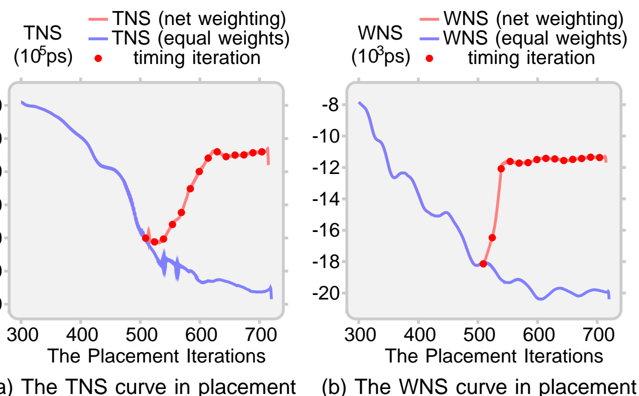
Benchmark	DREAMPlace [26]		DREAMPlace [26]+ [10]		OpenROAD [37]		Our GP [1]		Our GP + Our DP	
	HPWL	Runtime	HPWL	Runtime	HPWL	Runtime	HPWL	Runtime	HPWL	Runtime
superblue1	420.3	164.69	426.6	1320.73	462.4	8986.78	443.1	977.56	446.5	2242.98
superblue3	474.3	153.93	480.9	1247.24	598.7	17535.67	482.4	952.11	490.6	1866.07
superblue4	313.9	112.33	318.9	910.77	324.9	6296.16	335.9	610.26	346.4	2040.10
superblue5	492.5	202.87	571.9	1758.97	518.8	10882.33	556.2	1343.46	560.2	2490.95
superblue7	599.1	249.32	607.9	1968.70	627.4	11443.82	604.0	1537.42	620.8	3113.60
superblue10	935.9	308.81	941.8	1871.55	946.6	11467.05	1036.7	1288.63	1044.1	4485.27
superblue16	432.2	102.88	455.6	875.13	455.5	8861.90	448.1	542.15	495.0	1961.69
superblue18	232.9	104.06	240.1	887.29	264.8	4326.45	253.6	657.47	266.1	1457.67
Average Ratio	0.913	0.071	0.948	0.553	0.994	4.322	0.970	0.412	1.000	1.000

cof_{max} value (default to 1.05) until the global placement converged. The cof_{max} values in our experiments on the ICCAD 2015 contest benchmarks [25] are set to 1.02, 1.01, 1.02, 1.02, 1.05, 1.05, 1.015, and 1.05 for the eight benchmarks in order. More details of the physical meanings of cof_{max} can be found in the experiment section of RePlAce [39]. In addition, we particularly set the stop overflow threshold (default to 0.1) of superblue5 to 0.125 as there exists convergence difficulty when OpenROAD [37] is trying to reduce the overflow. Compared to OpenROAD [37], our proposed algorithm in GP and DP shows competivity on the two main timing objectives TNS and WNS.

Compared to our previous work which only implement optimization at global placement stage, our additional detailed placement algorithm achieves further timing optimization on both TNS and WNS. Although our net weighting scheme at global placement stage outperforms other algorithms in an average sense, it is still inferior in some benchmark to [10]. And after detailed placement, our algorithm achieves optimality in most benchmarks.

C. Visualization in Global Placement

To visualize the impact of net weighting on TNS and WNS at global placement stage, we plot the TNS and WNS values on superblue18 after the 300th iteration in Fig. 6. At the beginning, the cells kept repelling each other, thus increases the wirelength while decreases TNS and WNS. The blue curves indicate the result without any timing-aware optimization, and the red ones shows how the objective function oscillate under the influence of net weighting. The timing iterations are emphasized with scattered red squares.



(a) The TNS curve in placement (b) The WNS curve in placement

Fig. 6 The TNS and WNS values at each global placement iteration after the 300th iteration for superblue18.

- At almost every timing iteration, marked in red in Fig. 6, TNS can be improved immediately, especially when the balance of net weights starts to break down.
- After one or two net weighting steps, the WNS is quickly and significantly optimized. Thereafter, it remains almost stable in the later stages of global placement.

If our net weighting algorithm works for every net instead

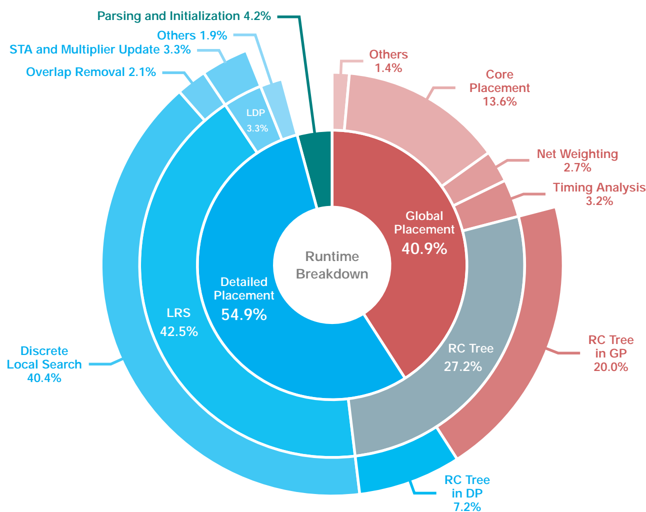


Fig. 7 The overall runtime breakdown on the ICCAD2015 contest benchmark superblue18 including both global and detailed placement.

of only some critical paths at a specific timing iteration, it makes sense when optimizing the TNS, which may contain many critical or near-critical paths. As for WNS, it may only provide information about few worst paths, which will be optimized quickly when the net weighting is applied for the first time. At later stages, other critical or nearly critical paths will be considered more often, and this is an important reason why it is difficult to optimize WNS further at global placement.

D. Wirelength and Runtime

We compare the circuit wirelength and runtime in TABLE III. The HPWL results in TABLE III are scaled from that evaluated by the ICCAD 2015 contest evaluation script. The site width is l_{site} is 380 in all benchmarks of [25] and the DEF unit u_{def} is 2000. The relationship between the evaluated HPWL W_{eval} and the reported HPWL W_{report} in TABLE III is determined by

$$W_{\text{report}} = \frac{10^{-6} u_{\text{def}}}{l_{\text{site}}} W_{\text{eval}}. \tag{29}$$

Net weighting targets at optimizing timing objectives, regardless of wirelength quality loss. One may explicitly set an upper bound to prevent net weights from becoming too large if required, so that the wirelength quality loss could be limited. TABLE III reveals that our proposed net weighting and detailed placement is still competitive compared to the outstanding timing-driven version of OpenROAD [37] on wirelength and runtime.

Since timing-driven placement has to perform STA and translate the feedback to certain operations, it significantly sacrifice runtime performance compared to DREAMPlace [26] which is extremely fast to optimize cell locations on GPUs.

Compared to [26] without any timing-aware optimization, we roughly take 5 times runtime to optimize negative slacks in global placement. The detailed placement takes more time to refine the solution as it has to re-calculate the timing delay for a huge amount of delay arcs. Although the runtime degradation posed by timing analysis is significant and inevitable, our entire timing-driven global optimization consumes an acceptable time compared to OpenROAD [37].

TABLE IV Comparison of TNS and WNS results with different cell delay models in detailed placement. The TNS unit and the WNS unit are 10^5ps and 10^3ps, respectively.

Benchmark	Timing model in [24]		Ours	
	TNS	WNS	TNS	WNS
superblue1	-66.4884	-13.1423	-67.0417	-13.0626
superblue3	-52.7973	-17.0728	-52.2784	-16.2643
superblue4	-136.366	-11.1400	-136.174	-11.0606
superblue5	-93.2448	-25.1624	-94.0022	-23.7035
superblue7	-57.5153	-15.2163	-58.3746	-15.2163
superblue10	-707.714	-26.1679	-705.795	-25.3786
superblue16	-119.500	-11.3635	-118.697	-11.2812
superblue18	-43.9558	-11.7931	-43.4910	-11.6939
Average Ratio	1.000	1.021	1.000	1.000

Fig. 7 plots the overall runtime breakdown on the benchmark superblue18 for both the global and detailed placement. At the global placement stage, constructing RC tree is the main bottleneck, which is accomplished on CPUs and thus very time-consuming, especially for large nets. Since STA must be called multiple times to incorporate changes of cell locations, the overhead of STA and RC tree construction should be the focus of the acceleration.

At the detailed placement stage, we are facing the runtime bottleneck dominated by the discrete local search, as we need to estimate the delay change for a number of location candidates. We set the window size to 63 5 and search stride to 5 site width, which reach a balance between runtime and placement quality. Note that our delay estimation in Section IV-A is based on the assumption that each node will not be moved too far away from the original location. A large window size setting will give too many candidate locations, resulting in large runtime overhead. Besides, a large displacement may incur significant inaccuracy of delay estimation. On the other hand, too small window size apparently would prevent us from finding better cell locations as the solution space is limited.

E. Detailed Placement Optimization

Since we adopt a more accurate cell delay model for cost computation in our detailed placement, we also compare the TNS and WNS results with different timing models in TABLE IV. The first two columns are results obtained using the rough linear delay model in [24]. It is confirmed in TABLE IV that a more accurate delay model can lead to a significant WNS improvement.

Differentiable timing-driven placement [40] is another powerful technique to optimize timing in global placement. The significant improvements of TNS and WNS can be observed as they are integrated as penalty terms in the general objective of differentiable timing [40]. The gradient computation of the

TABLE V Comparison of TNS, WNS, HPWL and Runtime before and after our detailed placement based on the results of differentiable-timing-driven placement [40]. The TNS unit and the WNS unit are 10^5ps and 10^3ps, respectively. The HPWL is evaluated by the ICCAD 2015 contest evaluation script and scaled according to Equation (29). The unit of Runtime is second.

Benchmark	[40]				[40] + Our DP			
	TNS	WNS	HPWL	Runtime	TNS	WNS	HPWL	Runtime
superblue1	-74.8538	-10.7695	423.8	268.31	-74.3638	-10.4355	424.0	1106.76
superblue3	-39.4299	-12.3742	478.4	266.65	-37.1255	-11.9170	479.1	1087.66
superblue4	-82.9243	-8.49154	312.2	156.36	-78.9090	-8.00840	313.3	1706.42
superblue5	-108.076	-25.2123	488.7	259.26	-102.491	-23.7488	491.1	1684.31
superblue7	-46.4264	-15.2163	602.1	450.85	-45.2301	-15.2163	602.5	1437.83
superblue10	-558.054	-21.9740	934.4	265.24	-515.003	-18.7072	936.6	2962.40
superblue16	-87.0255	-10.8544	485.1	217.65	-66.2820	-9.67800	491.0	1680.10
superblue18	-19.3143	-7.98730	243.6	156.99	-16.6570	-5.75639	243.9	968.39
Average Ratio	1.095	1.110	0.997	0.178	1.000	1.000	1.000	1.000

timing objectives in [40] is strongly bonded to the delay model and directly implemented according to an explicit form of gradients. Therefore, this approach is limited by the adopted timing model, resulting in unavoidable inflexibility.

TABLE V indicates that our detailed placement is able to further optimize TNS and WNS based on the results of the differentiable-timing-driven placement [40]. We acquired the placement solutions from the authors of [40] for the initialization of detailed placement. The slack improvement is up to 10% for both TNS and WNS on the ICCAD2015 contest benchmarks [25], which confirms the effectiveness of the detailed placement.

VI. Conclusion

In this paper, we propose a momentum-based net weighting scheme for timing-driven global placement, enhance the preconditioner accordingly and further improve timing quality in the timing-driven detailed placement based on Lagrangian multipliers. The evaluation results on ICCAD2015 contest benchmarks show that we can significantly improve both TNS and WNS. We use different strategies to optimize timing in these two stages, but both inspire us to notice the importance of placement in physical design. Although most timing-aware optimization methods are performed at incremental stages, it is still very effective to consider timing at the earlier stages of physical design, especially the global placement and the detailed placement.

References

[1] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin, and B. Yu, "DREAMPlace 4.0: timing-driven global placement with momentum-based net weighting," in *IEEE/ACM Proceedings Design, Automation and Test in Eurpoe (DATE)*. IEEE, 2022, pp. 939–944.

[2] I. L. Markov, J. Hu, and M.-C. Kim, "Progress and challenges in VLSI placement research," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 1985–2003, 2015.

[3] M. Burstein and M. N. Youssef, "Timing influenced layout design," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1985, pp. 124–130.

[4] A. E. Dunlop, V. D. Agrawal, D. N. Deutsch, M. Jukl, P. Kozak, and M. Wiesel, "Chip layout optimization using critical path weighting," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1984, pp. 133–136.

[5] H. Chang, E. Shragowitz, J. Liu, H. Youssef, B. Lu, and S. Sutanthavibul, "Net criticality revisited: An effective method to improve timing in physical design," in *ACM International Symposium on Physical Design (ISPD)*, 2002, pp. 155–160.

[6] T. Kong, "A novel net weighting algorithm for timing-driven placement," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2002, pp. 172–176.

[7] B. Halpin, C. R. Chen, and N. Sehgal, "A sensitivity based placer for standard cells," in *Proceedings of the 10th Great Lakes symposium on VLSI*, 2000, pp. 193–196.

[8] T.-Y. Wang, J.-L. Tsai, and C. C.-P. Chen, "Sensitivity guided net weighting for placement driven synthesis," in *ACM International Symposium on Physical Design (ISPD)*, 2004, pp. 124–131.

[9] Z. Xiu and R. A. Rutenbar, "Timing-driven placement by grid-warping," in *ACM/IEEE Design Automation Conference (DAC)*, 2005, pp. 585–591.

[10] H. Eisenmann and F. M. Johannes, "Generic global placement and floorplanning," in *ACM/IEEE Design Automation Conference (DAC)*, 1998, pp. 269–274.

[11] B. M. Riess and G. G. Ettelt, "Speed: Fast and efficient timing driven placement," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1. IEEE, 1995, pp. 377–380.

[12] B. Obermeier and F. M. Johannes, "Quadratic placement using an improved timing model," in *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2004, pp. 705–710.

[13] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[14] W. K. Luk, "A fast physical constraint generator for timing driven layout," in *ACM/IEEE Design Automation Conference (DAC)*, 1991, pp. 626–631.

[15] T. Gao, P. M. Vaidya, and C. Liu, "A Performance Driven Macro-Cell Placement Algorithm." in *ACM/IEEE Design Automation Conference (DAC)*, 1992, pp. 147–152.

[16] R.-S. Tsay and J. Koehl, "An analytic net weighting approach for performance optimization in circuit placement," in *ACM/IEEE Design Automation Conference (DAC)*, 1991, pp. 620–625.

[17] K. Rajagopal, T. Shaked, Y. Parasuram, T. Cao, A. Chowdhary, and B. Halpin, "Timing driven force directed placement with physical net constraints," in *ACM International Symposium on Physical Design (ISPD)*, 2003, pp. 60–66.

[18] A. Chowdhary, K. Rajagopal, S. Venkatesan, T. Cao, V. Tiourin, Y. Parasuram, and B. Halpin, "How accurately can we model timing in a placement engine?" in *ACM/IEEE Design Automation Conference (DAC)*, 2005, pp. 801–806.

[19] M. A. Jackson and E. S. Kuh, "Performance-driven placement of cell based IC's," in *ACM/IEEE Design Automation Conference (DAC)*, 1989, pp. 370–375.

[20] W. Swartz and C. Sechen, "Timing driven placement for large standard cell circuits," in *ACM/IEEE Design Automation Conference (DAC)*, 1995, pp. 211–215.

[21] T. Hamada, C.-K. Cheng, and P. M. Chau, "Prime: A timing-driven placement tool using a piecewise linear resistive network approach," in *ACM/IEEE Design Automation Conference (DAC)*, 1993, pp. 531–536.

[22] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," vol. 323, no. 6088, pp. 533–536, 1986.

[23] J. Lu, P. Chen, C.-C. Chang, L. Sha, D. J.-H. Huang, C.-C. Teng, and C.-K. Cheng, "ePlace: Electrostatics-based placement using fast fourier transform and Nesterov's method," vol. 20, no. 2, pp. 1–34, 2015.