DOI:10.1145/2461256.2461279

SimPL: An Algorithm for Placing VLSI Circuits

By Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov

Abstract

VLSI placement optimizes locations of circuit components so as to reduce interconnect. Formulated in terms of (hyper) graphs, it is NP-hard, and yet must be solved for challenging million-node instances within several hours. We propose an algorithm for large-scale placement that outperforms prior art both in runtime and solution quality on standard benchmarks. The algorithm is more straightforward than existing placers and easier to integrate into timing-closure flows. Our C++ implementation is compact, self-contained and exploits instruction-level and thread-level parallelism. Due to its simplicity and superior performance, the algorithm has been adopted in the industry and was extended by several university groups to multi-objective optimization.

1. INTRODUCTION

The first algorithms for circuit placement have been developed at Bell Labs and IBM Research in the 1960s and followed the divide-and-conquer paradigm. They motivated high-performance heuristics for balanced graph-partitioning by Kernighan and Lin and, later, by Fiduccia and Mattheyses, that minimize edge cut. In the mid-1980s, circuit placement was a key application of the newly invented Simulated Annealing methods. Fifteen years later, the number of components in leading chips grew to the point where annealing was much too slow. The divide-and-conquer framework temporarily regained leadership when it was combined with bottom-up clustering and multi-level partitioning. However, in the 2000s, increasing transistor density again demanded faster algorithms with better performance. Linear programming and network flows were tried with limited success.

Placement optimization gradually became more significant in chip design over the years because the amount of interconnect grows faster than the number of components (except for grid-like circuits such as memory blocks). On-chip interconnect now occupies greater volume than transistors and consumes much power. Additionally, transistor delays improve faster than interconnect delay, which today limits the speed of many chips. This is why circuit placement has recently been integrated with more comprehensive optimizations that can reduce interconnect by restructuring the circuit.[1] But such optimizations need initial component locations that minimize edge lengths. This puts an easy-to-formulate graph problem at the core of sophisticated industrial optimizations. For details the readers are referred to Chapters 4 and 8 of Kahng et al.[12]

Modern techniques for VLSI placement approximate interconnect length by differentiable functions and draw on efficient numerical optimizations. Such *global* placement tolerates various geometric misalignments and small overlaps between rectangular components (represented by graph nodes), which are subsequently repaired by combinatorial algorithms for *legalization* and *detailed* placement. Despite impressive improvements reported by researchers[15] and industry software in the last decade, global-placement algorithms suffer several key shortcomings: (*i*) speed, (*ii*) solution quality, (*iii*) simplicity and integration with other optimizations, and (*iv*) support for multi-threaded execution.

State-of-the-art algorithms for global placement form two families: (*i*) *force-directed* quadratic placers, such as Kraftwerk2,[20] FastPlace3,[22] and RQL,[23] and (*ii*) *nonconvex optimization* techniques, such as APlace2,[8] NTU-Place3,[4] and mPL6.[3] To form an intuition about force-directed algorithms, one thinks of individual interconnects as coil springs subject to Hooke's law and seeks a force-equilibrium (min-energy) configuration. Mathematically, the total interconnect length is captured by a quadratic function of component locations and minimized by solving a large sparse system of linear equations. To discourage component overlap, *forces* are added by pulling components away from high-density areas. These forces are represented by *pseudonodes* and *pseudoedges*, which extend the original quadratic function.[7] They are updated after each linear-system solve until iterations converge. Nonconvex optimization models interconnect length by more sophisticated differentiable functions that grow linearly with length. These functions are minimized by the nonlinear conjugate gradient method. Component density is modeled by functional terms, which are more accurate than forces, but also requires updates after each change to placement.[4, 8] Algorithms in both categories are used in the industry or closely resemble those in industry placers.

Nonconvex optimization methods previously claimed the best results for academic implementations[4] and industry software, but are significantly slower, which is problematic for modern chip designs with components in many millions. To scale the basic nonconvex optimization framework, best tools in this family employ *hypergraph clustering* and *multilevel/multigrid extensions*, sometimes at the cost of solution quality. Such multilevel placers perform many sequential steps, obstructing efficient parallelization. Moreover, clustering and refinement do not fully benefit from modern multicore CPUs. Owing to their complexity, multilevel placers are also harder to maintain and combine with other optimizations. In particular, clustered circuits obscure analysis

The original version of this paper appeared in the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (Jan. 2012)

of routing congestion and timing, and complicate circuit restructuring. State-of-the-art force-directed quadratic placers tend to run many times faster than nonconvex optimization, but also use multilevel extensions in their most competitive configurations. Their solution quality is mixed.

In this work, we develop a self-contained technique for global placement based on quadratic programming. It maintains lower-bound and upper-bound placements that converge to a final solution. The upper-bound placement is produced by our new feasibility projection algorithm based on top-down geometric partitioning and nonlinear scaling. Research in VLSI placement includes a fiercely competitive benchmarking component, and we show that our algorithm performs very well on standard benchmarks.

In the remainder of this paper, Section 2 describes the building blocks from which our algorithm was assembled. Section 3 introduces our key ideas and articulates our solution of the *force modulation* problem. The SimPL algorithm is presented in Section 4 along with complexity analysis. Empirical validation is described in Section 5. The use of parallelism is discussed in Section 6.

2. ESSENTIAL CONCEPTS

Circuit placement typically operates on a gate-level netlist, which consists of standard cells (NAND, NOR, MUX, half-adders, etc.) and interconnect. Each standard cell has a rectangular footprint with well-defined area. A cell's output may connect to inputs of multiple other cells—such interconnects are captured by hyperedges, also known as *signal nets*. Given a netlist $\mathcal{N} = (E, V)$ with nets E and nodes (cells) V, *global placement* seeks node locations (x_i, y_i) such that the area of nodes within any rectangular region does not exceed the area of (cell sites in) that region.[a] Some locations of cells may be given initially and fixed. The interconnect objective optimized by global placement is the Half-Perimeter WireLength (HPWL). While easy to calculate, HPWL is a surprisingly good estimate of the length of routed connections. For node locations $\vec{x} = \{x_i\}$ and $\vec{y} = \{y_i\}$, $\mathrm{HPWL}_{\mathcal{N}}(\vec{x}, \vec{y}) = \mathrm{HPWL}_{\mathcal{N}}(\vec{x}) + \mathrm{HPWL}_{\mathcal{N}}(\vec{y})$, where

$$HPWL_{\mathcal{N}}(\vec{x}) = \sum_{e \in E} \left[\max_{i \in e} x_i - \min_{i \in e} x_i \right] \quad (1)$$

This formula generalizes the so-called Manhattan (taxicab) distance between two points. Given the rigorous public benchmarking infrastructure developed by IBM Research and academic colleagues,[15] consistent improvements by even several percent are considered significant in both academic literature and industry practice. Efficient optimization algorithms approximate $\mathrm{HPWL}_{\mathcal{N}}$ by differentiable functions.

Quadratic optimization. Consider a graph $\mathcal{G} = (E_{\mathcal{G}}, V)$ with edges $E_{\mathcal{G}}$, vertices V, and edge weights $w_{ij} > 0$ for all edges $e_{ij} \in E_{\mathcal{G}}$. The *quadratic objective* $\Phi_{\mathcal{G}}$ is defined as

$$\Phi_{\mathcal{G}}(\vec{x}, \vec{y}) = \sum_{i,j} w_{i,j} \left[(x_i - x_j)^2 + (y_i - y_j)^2 \right] \quad (2)$$

a In practice, this constraint is enforced for bins of a regular grid. The layout area is subdivided into equal, disjoint, small rectangles, so as to limit the area of cells placed inside.

Its x and y components are cast in matrix form[2, 20]

$$\Phi_{\mathcal{G}}(\vec{x}) = \frac{1}{2} \vec{x}^T Q_x \vec{x} + \vec{c}_x^T \vec{x} + \text{const} \quad (3)$$

The Hessian matrix Q_x captures connections between pairs of movable vertices, while vector $\vec{c}_x$ captures connections between movable and fixed vertices. For more details, the readers are referred to Section 4.3.2 of Kahng et al.[12] When Q_x is nondegenerate, $\Phi_{\mathcal{G}}(\vec{x})$ is a strictly convex function with a unique minimum, which can be found by solving the system of linear equations $Q_x \vec{x} = -\vec{c}_x$. Solutions can be quickly approximated by iterative Krylov-subspace techniques, such as the conjugate gradient (CG) method and its variants.[19] Since Q_x is symmetric positive definite, CG iterations provably minimize the residual norm. The convergence is monotonic,[21] but its rate depends on the spectral properties of Q_x, which can be enhanced by *preconditioning*. In other words, we solve the equivalent system $P^{-1}Q_x = -P^{-1}\vec{c}_x$ for a nondegenerate matrix P, such that P^{-1} is an easy-to-compute approximation of Q_x^{-1}. Given that Q_x is diagonally dominant, we chose P to be its diagonal, also known as the *Jacobi preconditioner*. We deliberately enhance diagonal dominance in Q_x (Section 4.3).

Quadratic placement example. Consider the graph $\mathcal{G}$ and edge weights w_{ij} in Figure 1. Quadratic placement minimizes the separable quadratic cost function $\Phi_{\mathcal{G}}$ in the x and y directions. For the x-direction,

$$\Phi_{\mathcal{G}}(\vec{x}) = w_{12}(x_1 - x_2)^2 + w_{13}(x_1 - x_3)^2 + w_{1f_1}(x_1 - f_1)^2 + w_{23}(x_2 - x_3)^2 + w_{34}(x_3 - x_4)^2 + w_{4f_2}(x_4 - f_2)^2$$

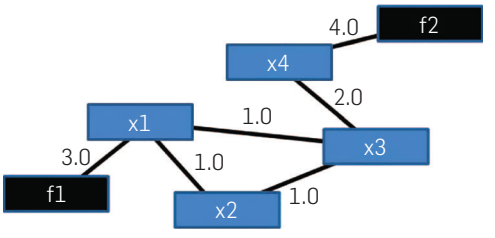
Setting the partial derivatives to 0 (the condition for force equilibrium), we solve for the global minimum cost.

$$\frac{\partial \Phi_{\mathcal{G}}(\vec{x})}{\partial x} = 0 \Leftrightarrow Q_x \vec{x} = -\vec{c}_x \Leftrightarrow \quad (4)$$

$$\begin{bmatrix} 5.0 & -1.0 & -1.0 & 0.0 \\ -1.0 & 2.0 & -1.0 & 0.0 \\ -1.0 & -1.0 & 4.0 & -2.0 \\ 0.0 & 0.0 & -2.0 & 6.0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 3.0 f_1 \\ 0.0 \\ 0.0 \\ 4.0 f_2 \end{bmatrix}$$

The connectivity matrix Q_x has entry w_{ij} in the ith row and jth column, and $-\vec{c}_x$ has entry c_i in the ith row. The diagonal entries w_{ii} correspond to the sum of net weights of all connections to movable module i. The off-diagonal entries w_{ij} are calculated as the negative sum of net weights of connections between movable modules i and j, and the resulting connectivity matrix becomes symmetric. Each element c_x for

Figure 1. Blue boxes represent movable modules and black boxes represent fixed modules.



a movable module i is calculated as the sum of $w_{ij} \cdot x_j$, where x_j is the pin location of each connected fixed module.With $(f_1, f_2) = (1.0, 3.5)$, a linear system solver finds a unique solution $\vec{x} = [1.4762\ 1.9524\ 2.4286\ 3.1429]^T$ that minimizes the quadratic wirelength $\Phi_{\mathcal{G}}(\vec{x})$.

The Bound2Bound net model.[20] To represent the HPWL objective by the quadratic objective, the netlist $\mathcal{N}$ is transformed into *two* graphs, $\mathcal{G}_x$ and $\mathcal{G}_y$, that preserve the node set V and represent each two-pin net by a single edge with weight $1/length$. Larger nets are decomposed depending on node locations—for each p-pin net, the *extreme* nodes (min and max) are connected to each other and to each *internal* node by edges, with the following weight

$$w_{x,ij}^{B2B} = \frac{1}{(p-1)|x_i - x_j|} \quad (5)$$

For example, 3-pin nets are decomposed into cliques[14] with edge weight $1/2l$, where l is the length of a given edge. In general, this quadratic objective and the Bound2Bound (B2B) net decomposition capture the HPWL objective exactly, but only for the given placement. As locations change, the approximation error may grow, necessitating multiple updates throughout the placement algorithm.

Most quadratic placers use the placement-independent star or clique decompositions, so as not to rebuild Q_x and Q_y many times.[2, 22, 23] Yet, the B2B model uses fewer edges than cliques ($p > 3$), avoids new variables used in stars, and is more accurate than both stars and cliques.[20]

3. KEY IDEAS IN OUR WORK

Analytic placement techniques first minimize a function of interconnect length, neglecting overlaps between standard cells and macros. This initial step places many cells in densely populated regions, typically around the center of the layout. Cell locations are then gradually spread through a series of placement iterations, during which interconnect length slowly *increases*, converging to a final overlap-free placement (a small amount of overlap is often allowed and later resolved during legalization).

Our algorithm also starts with interconnect minimization, but its next step is unusual—most overlaps are removed using a fast *look-ahead legalizer* based on top-down geometric partitioning and nonlinear scaling. Locations of movable objects in the legalized placement serve as *anchors* that coerce the initial locations to reduce overlap by adding pseudonets to baseline force-directed placement.[7] Each subsequent iteration of our algorithm produces (*i*) an almost-legal placement that *overestimates* the final result through look-ahead legalization and (*ii*) an illegal placement that *underestimates* the final result—through linear system solver. The wirelength gap between lower- and upper-bound placements helps monitor convergence (Section 4.3).

Solving the force-modulation problem. A key innovation in SimPL is the interaction between the lower-bound and the upper-bound placements—it ensures convergence to a no-overlap solution while optimizing interconnect length. It solves two well-known challenges in analytic placement: (1) finding directions in which to spread the locations (*force orientation*) and (2) determining the appropriate amount of spreading (*force modulation*).[13, 23] This is unlike previous work, where spreading directions are typically based on *local information*, for example, placers based on nonconvex optimization use *gradient* information and require a large number of expensive iterations. Kraftwerk2[20] orients spreading forces according to solutions of Poisson's equation, providing a global perspective and speeding up convergence. However, this approach does not solve the force-modulation problem, as articulated in Kennings and Vorwerk.[13] The authors of RQL,[23] which can be viewed as an improvement on FastPlace, revisit the force-modulation problem and address it by a somewhat *ad hoc* limit on the magnitude of spreading forces. In our work, *look-ahead legalization algorithm* (Section 4.2), invoked at each iteration, determines both the direction and the magnitude of spreading forces. It is global in nature, accounts for fixed obstacles, and preserves relative placement to ensure interconnect optimization and convergence.

Global placement with look-ahead. The legalized upper-bound placements built at every iteration can be viewed as *look-ahead* because they are used temporarily and not refined directly. The look-ahead placements approximately satisfy constraints (e.g., legality and placement density) while trying to retain quality of current lower-bound placements as much as possible. These locations are then used to update the current lower-bound placements by evolving them toward look-ahead placements. They pull cell locations in lower-bound placements not just away from dense regions but also toward the regions where space is available. Such *area look-ahead* is particularly useful around fixed obstacles, where local information does not offer sufficient guidance. Similar *congestion look-ahead*,[6, 11] *power look-ahead, thermal look-ahead*, and *timing look-ahead* based on legalized placements help integrate our placement algorithm into multi-objective circuit optimizations.

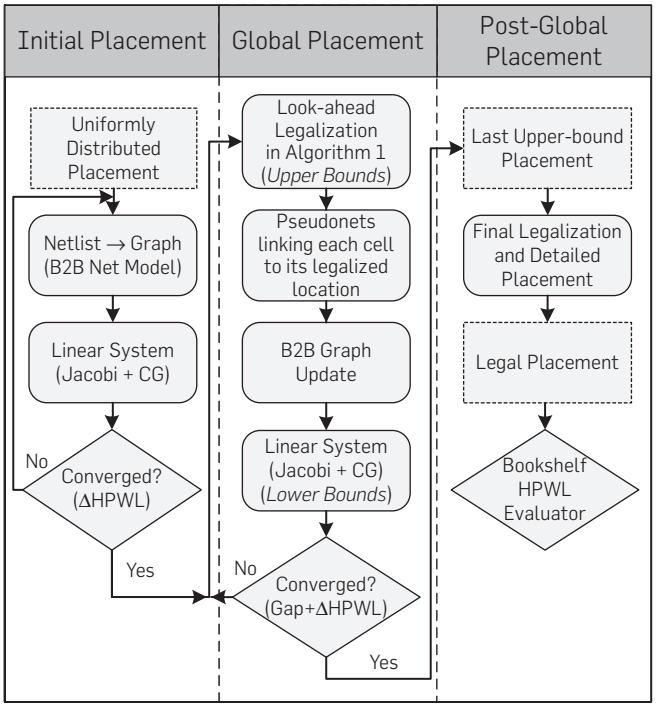
4. OUR GLOBAL PLACEMENT ALGORITHM

Our placement technique consists of three phases: initial placement, global placement iterations, and post-global placement (Figure 2). Initial placement, described next, is mostly an exercise in judicious application of known components. Our main innovation is in the global placement phase. Post-global placement (legalization and detailed placement) is straightforward, given current state of the art.

4.1. Initial placement

Our initial-placement step is conceptually similar to those of other force-directed placers[20, 22, 23]—it entirely ignores cell areas and overlaps, so as to minimize the objective function, a quadratic approximation of total interconnect length. We found that this step notably impacts the final result, as it can determine the overall shape of the final placement solutions. Therefore, unlike FastPlace3[22] and RQL,[23] we use the more accurate B2B net model from Spindler et al.[20] reviewed in Section 2. After the first quadratic solve, we rebuild the circuit graph because the B2B net model is placement-dependent. We then alternate quadratic solves and graph rebuilding until HPWL stops improving. In practice, this requires a small number of

Figure 2. The SimPL algorithm uses placement-dependent B2B net model, which is updated on every iteration. *Gap* refers to the difference between upper and lower bounds.



Algorithm 1. Look-ahead Legalization by Top-down Geometric Partitioning and Nonlinear Scaling.

Maximum allowed density γ, where $0 < \gamma < 1$
Placement of cells
Queue of bin clusters $Q = 0$

1: Identify γ-overfilled bins and cluster them // Figure 3(a)
2: **foreach** cluster c **do**
3: Find a minimal rectangular region $R \supset c$ with density$(R) \leq \gamma$
4: $R.level = 1$
5: Q.enqueue(R)
6: **while** !Q.empty() **do**
7: $B = Q$.dequeue()
8: **if** (Area(B) is small enough $\|$ $B.level \geq 10$) **then**
9: **continue**
10: $M = \{$movable cells in $B\}$
11: C_c = A cutline to evenly split cell area in M
12: C_B = A cutline to evenly partition whitespace in B
13: $(S_0, S_1) = \{$two sub-regions of B created by cutline $C_c\}$
14: $(M_0, M_1) = \{$movable cells in $S_0, S_1\}$
15: $(B_0, B_1) = \{$two sub-regions of B created by cutline $C_B\}$
16: Perform NONLINEAR SCALING on $M_0(M_1)$ in $B_0(B_1)$
17: $B_0.level = B_1.level = B.level + 1$
18: Q.enqueue(B_0, B_1)
19: **end while**
20: **end foreach**

iterations (5–7), regardless of benchmark size, because the relative ordering of locations stabilizes quickly.

4.2. Look-ahead legalization

Consider a set of cell locations produced by quadratic optimization (lower-bound placement) with a significant amount of overlap as measured using bins of a regular grid. Look-ahead legalization is invoked at each iteration of global-placement process to change the global positioning of those locations, seeking to remove most of the overlap (with respect to the grid) while preserving the relative ordering.[b] This step can be viewed as a projection of the lower-bound placement onto the manifold of feasible placements. The quality of look-ahead legalization is measured by its impact on the entire placement flow. Our look-ahead legalization is based on top-down recursive geometric partitioning and nonlinear scaling (Algorithm 1). Cutlines C_c and C_B are chosen to be vertical at the top level ($R.level = 1$), and they alternate between horizontal and vertical directions with each successive level of top-down geometric partitioning.

Handling density constraints. For each grid bin of a given regular grid, we calculate the total area of contained cells A_c and the total available area of cell sites A_a. A bin is γ-overfilled if its *cell density* A_c/A_a exceeds given density limit $0 < \gamma < 1$. Adjacent γ-overfilled bins are clustered by Breadth-First Search (BFS), and look-ahead legalization is performed on such clusters. For each cluster, we find a minimal containing rectangular region with density $\leq \gamma$ (these regions can also be referred to as "clusters"). A key insight is that overlap removal in a region, which is filled to capacity, is more straightforward because the absence of whitespace leaves less flexibility for interconnect optimization.[c] If relative placement must be preserved, overlap can be reduced by means of x- and y-sorting with subsequent greedy packing. The next step, *nonlinear scaling*, implements this intuition, but relies on cell-area cutline C_c chosen in Algorithm 1 and shifts it toward the median of available area C_B in the region, so as to equalize densities in the two sub regions (Figure 3).

Nonlinear scaling in one direction is illustrated in Figure 4, where a new region was created by a vertical cutline C_B during top-down geometric partitioning. This region is subdivided into vertical stripes parallel to C_B. First, cutlines are drawn along the boundaries of obstacles present in this region. Each vertical stripe created in this process is further subdivided if its available area exceeds 1/10 of the region's available area. Movable cells in the corresponding sub-region created by C_c are then sorted by their distance from C_B and greedily packed into the stripes in that order. In other words, the cell furthest from the cutline is assigned to the furthest stripe. Each subsequent cell is assigned to the furthest unfilled stripe. For each stripe, we calculate the available site area A_a and consider the stripe filled when the area

b This formulation is related to the Monge–Kantorovich optimal transport, although in our context runtime is extremely limited and optimal solutions are not required.

c In the presence of whitespace, the placer can move cells around without changing their relative ordering. Removing whitespace suppresses this degree of freedom, giving fewer choices to the placer.

Figure 3. Clustering of overfilled bins in Algorithm 1 and adjustment of cell-area to whitespace median by nonlinear scaling (also see Figure 4). Movable cells are shown in blue, obstacles in solid gray.

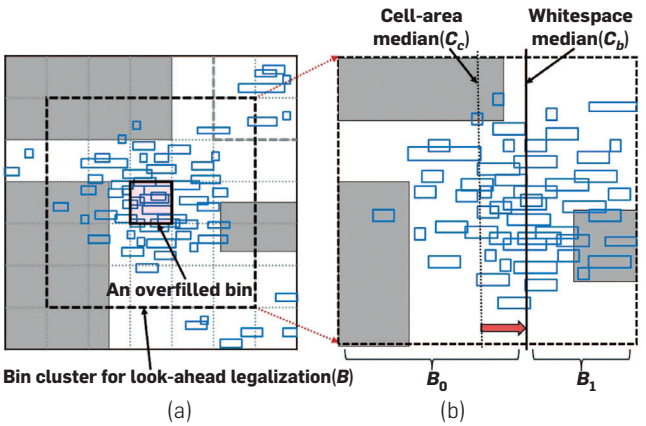
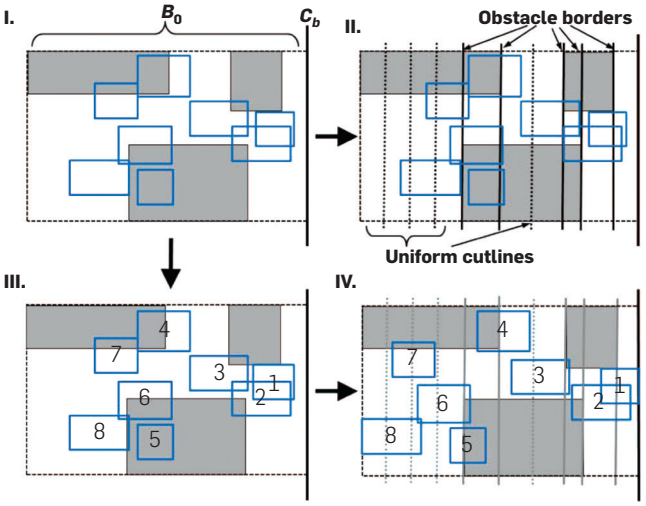


Figure 4. Nonlinear scaling in a region with obstacles (I): the formation of C_B-aligned stripes (II), cell sorting by distance from C_B (III), and greedy cell positioning (IV).



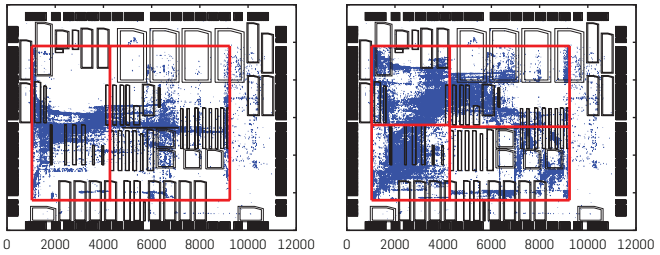
of assigned cells reaches γA_a. Cell locations within each stripe are linearly scaled from current locations (nonlinearity arises from different scaling in different stripes).

Look-ahead legalization applies nonlinear scaling in alternating directions, as illustrated in Figure 5 on one of ISPD 2005 benchmarks. Here, a region R is selected that contains overfilled bins, but is wide enough to facilitate overlap removal. R is first partitioned by a vertical cutline, after which nonlinear scaling is applied in the two new subregions. Subsequently, look-ahead legalization (Algorithm 1) considers each sub region individually and selects different horizontal cutlines. Four rounds of nonlinear scaling follow, spreading cells over the region's expanse (Figure 5).

4.3. Global placement iterations

Using legalized locations as anchors. Solving an unconstrained linear system results in a placement with

Figure 5. Nonlinear scaling after the first vertical cut and two subsequent horizontal cuts (ADAPTEC1) between iterations 0 and 1 in Figure 8.



significant amount of overlap. To pull cells away from their initial positions, we gradually perturb the linear system. As explained in Section 4.2, at each iteration of our global placement, top-down geometric partitioning and nonlinear scaling generate a roughly legalized solution. We use these legalized locations as fixed, zero-area anchors connected to their corresponding cells in the lower-bound placement with artificial two-pin pseudonets. Furthermore, following the discussion in Section 2, we note that connections to fixed locations do not increase the size of the Hessian matrix Q, and only contribute to its diagonal elements. For more details, the readers are referred to Section 4.3.2 of. Kahng et al.[12] This enhances diagonal dominance, condition number of $P^{-1}Q$, and the convergence rate of Jacobi-preconditioned CG.

In addition to weights given by the B2B net model on pseudonets, we control cell movement and iteration convergence by multiplying each pseudonet weight by an additional factor $\alpha > 0$ computed as $\alpha = 0.01 \times (1 + \text{Iteration\_Number})$.[d] At early iterations, small α values weaken spreading forces, giving greater significance to interconnect and more freedom to the linear system solver. As the relative ordering of cells stabilizes, increasing α values boost the pull toward the anchors and accelerate the convergence of lower bounds and upper bounds. Mathematically, the α parameter can be viewed as a Lagrange multiplier. The relevant constraint requires that each cell be placed over its anchor, and the (Manhattan) distance between their locations is the penalty for violating the constraint. The α parameter gradually increases and shifts the emphasis of quadratic optimization from reducing interconnect to satisfying constraints (Figure 6).

Convergence criteria similar to that in Section 4.1 can be adopted in global placement. We alternate (1) look-ahead legalization, (2) updates to anchors and the B2B net model, and (3) solution of the linear system, until HPWL of solutions generated by look-ahead legalization stops improving. Unlike in the initial placement step, however, HPWL values of upper-bound solutions oscillate during the first four to seven iterations, as seen in Figure 7. To prevent premature termination, we monitor the gap between the lower and upper bounds. Global placement continues until (1) the gap is reduced to 25% of the gap at the 10th iteration and upper-bound solution stops improving or (2) the gap is smaller

d Further improvements in pseudonet weighting and convergence are proposed in Kim and Markov.[9]

Figure 6. An anchor with a pseudonet. The α parameter prices the penalty for the cell being far from its anchor.

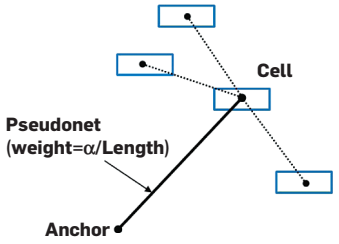
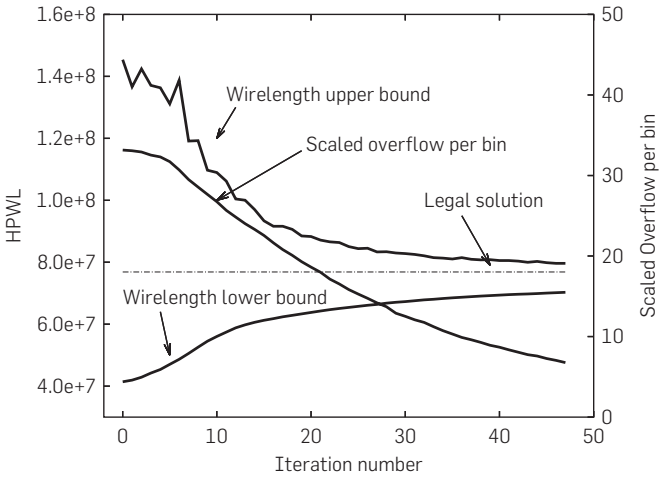


Figure 7. Lower and upper bounds for HPWL, the scaled overflow per bin (a placement density metric) of the lower-bound placement at each iteration, and HPWL of the legal placement (adaptec1).



than 10% of the gap at the 10th iteration. On the ISPD 2005 benchmark suite, only 33–45 iterations are needed. The final set of locations (global placement) is produced by the last look-ahead legalization, as shown in Figure 2.

Convergence is guaranteed by the increasing weights of pseudonets. At each iteration, these pseudonets pull the lower-bound placement toward a legalized upper-bound placement. As the lower-bound placement becomes closer to a legal placement, it exhibits a decreasing amount of cell overlap. This, in turn, results in smaller cell displacements during look-ahead legalization. After the first few iterations, one typically observes monotonic convergence (see Figure 7). A progression of global placement is annotated with HPWL values in Figure 8.

4.4. Asymptotic complexity analysis

The runtime of global placement iterations is dominated by the conjugate gradient (CG) solver and look-ahead legalization. The complexity of each CG invocation is $O\left(m\sqrt{\kappa}\right)$, where κ is the conditioning number of the matrix and m is the number of nonzero elements.[21] The number of nonzeros reflects the number of graph edges in the B2B model of the netlist. It grows linearly with the number of *pins* (cell-to-net connections)—a key size metric of a netlist. Another way to estimate the number of nonzeros is to observe that the average cell degree (the number of nets connected to a cell) is bounded by $d = 5$, or perhaps a slightly larger constant, for practical netlists. Since $m \leq (d+1)n$ for n cells (including diagonal elements), CG runs in $O\left(n\sqrt{\kappa}\right)$ time.

Asymptotic runtime of look-ahead legalization is dominated by sorting cell locations by their x and y coordinates because nonlinear scaling takes $O(n)$ time (several other linear-time steps take even less time in practice, therefore we do not discuss them). Given that look-ahead legalization operates on blocks of progressively smaller size, we can separately consider its processing pass for the top-level blocks, then the pass for half-sized blocks, etc. Only $O(\log n)$ such passes are required for n cells. Each pass takes $O(n \log n)$ time because top-level blocks do not experience significant overlaps—in fact, each subsequent pass becomes faster because sorting is applied to smaller groups of cells. Hence, look-ahead legalization runs in $O(n \log^2 n)$ time.

We have observed that owing to preconditioning, iteration counts in CG grow no faster than $\log n$, and each iteration takes linear time in n. Therefore, one global placement iteration takes $O(n \log^2 n)$ time. In practice, SimPL requires less than 50 placement iterations, even for million-gate circuits.

5. EMPIRICAL VALIDATION

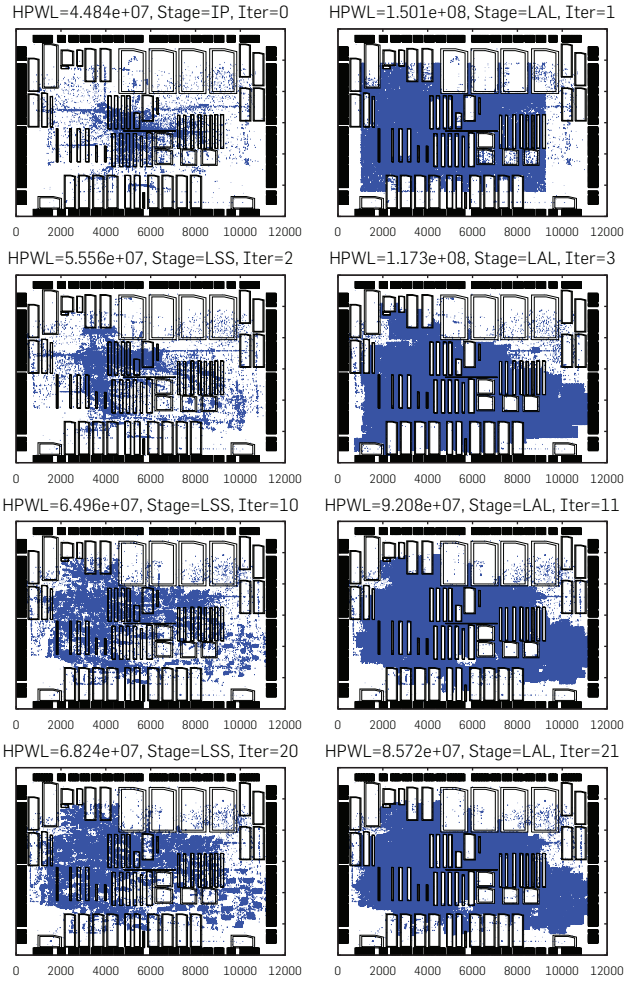
The SimPL global placer is implemented as a stand-alone software package with self-contained I/O, and initial placement and global placement iterations. Living up to its name, it consists of fewer than 5000 lines of C++ code and relies only on standard C++ libraries (shipped with g++ 4.4.0). Single-threaded benchmark runs were performed on an Intel Xeon Quad CPU E31230 (3.2 GHz) Linux workstation with 8GB RAM. We compared SimPL to other academic placers on the ISPD 2005 placement contest benchmark suite[e] with target density $\gamma = 1.0$. Focusing on global placement, we delegate final legalization (into rows and sites) and detailed placement to FastPlace-DP.[16]

Running in a single thread, SimPL completes the entire ISPD 2005 benchmark suite in 1 hour 3 minutes, placing the largest benchmark, BIGBLUE4 (2.18 M cells), in 33 minutes using 2.1GB of memory. We report the runtime breakdown on BIGBLUE4 according to Figure 2, excluding 1.4% runtime for I/O. *Initial placement* takes 5.0% of total runtime, of which 3.7% is spent in CG, and 1.3% in building B2B net models and sparse matrices for CG. *Global placement iterations* take 47.4%, of which 19% is in the CG solver, and 9.9% is in sparse matrix construction and B2B net modeling. Look-ahead legalization takes 17.7%. *Legalization and detailed placement* take 46.2%.

When compared to prior software for VLSI placement (Table 1), SimPL found placements with the lowest interconnect length and was the fastest. On average, SimPL obtains wirelength improvement of 16.26%, 4.12%, 4.23%, and 2.57% versus Capo10.5,[18] NTUPlace3,[4] FastPlace3,[22] and mPL6,[3] respectively. In comparison, one step of Moore scaling reduces interconnect by 30% at the cost several billion dollars. SimPL was 7.28 times faster than mPL6,

e http://archive.sigda.org/ispd2005/contest.htm.

Figure 8. A progression of global placement snapshots from different iterations and algorithm steps (adaptec1). IP = Initial Placement, LAL = Look-ahead Legalization, LSS = Linear System Solver. Left-side placements show lower bounds and right-side placements show upper bounds.



which appears to be the strongest preexisting placer. SimPL was 1.12 times faster than FastPlace3—previously the fastest academic software. Multi-objective placers based on SimPL[6, 11] also demonstrate their consistent speed advantages over other state-of-the-art placers, and especially so on larger circuit instances.

6. EXPLOITING PARALLELISM

Further speed-up is possible for SimPL on workstations with multicore CPUs.

Algorithmic details. Runtime bottlenecks in the sequential variant of the SimPL algorithm (Section 5)—updates to the B2B net model and the CG solver—can be parallelized. Given that the B2B net model is separable, we process the x and y cases in parallel. When more than two cores are available, we split the nets of the netlist into equal groups that can be processed by multiple threads. To parallelize the CG solver, we applied a coarse-grain *row partitioning* scheme to the Hessian Matrix Q, where different blocks of rows are assigned to different threads using OpenMP.[5] A core operation in CG is the Sparse Matrix-Vector multiply (SpMxV). Memory bandwidth is a known bottleneck and becomes more critical when multiple cores access the main memory through a common bus. We reduce memory bandwidth demand of SpMxV by using the *CSR* (Compressed Sparse Row)[19] memory layout for the Hessian matrix Q.

Our implementation exploits streaming SIMD extensions level 2 (SSE2)[17] that perform several floating-point operations at once, especially in the conjugate gradient solver. In practice, the impact of parallelization depends on the relation between CPU speed and memory bandwidth.

```
// inner product of two float vectors x and y
float inner_prod(vector<float> &x, vector<float> &y)
{
 __m128 thread_acc[NUM_THREADS], X, Y;
 float temp[4], inner_product=0.0;
 int i;
 for(int j = 0; j < NUM_THREADS; j++)
   thread_acc[j]=_mm_setzero_ps();
 #pragma omp parallel for private(X,Y) lastprivate(i)
 ...
     schedule(static) ordered num_threads(NUM_THREADS)
 for (i=0; i <= x.size()-4; i+=4)
 {
   X = _mm_load_ps(&x[i]);
   Y = _mm_load_ps(&y[i]);
   thread_acc[omp_get_thread_num()] = ...
     _mm_add_ps(thread_acc[omp_get_thread_num()], ...
     _mm_mul_ps(X,Y));
 }
 for(int j = 1; j < NUM_THREADS; j++)
   thread_acc[0]=_mm_add_ps(thread_acc[0],thread_
   acc[j]);
 _mm_store_ps(temp, thread_acc[0]);
 inner_product = temp[0] + temp[1] + temp[2] + temp[3];
 for ( ; i < x.size(); i++)
   inner_product += x[i] * y[i];
 return inner_product;
}
```

Listing 1. Sample code for OpenMP and SSE2 parallelization for the inner-product operation.

After we parallelized the CG solver, look-ahead legalization became a bottleneck and needed to be parallelized as well. To this end, top-down partitioning generates an increasing number of sub-tasks of similar sizes which can be solved independently. Let Q_g be the global queue of bin cluster from Algorithm 1 and Q_i be the private queue of bin clusters of thread i. First, we statically assign initial bin clusters to available threads such that each thread has similar number of bin clusters to start. After each level of top-down geometric partitioning and nonlinear scaling in such a bin cluster, each thread generates two sub-clusters with similar numbers of cells. Then, thread t_i adds only one of two sub-clusters to its private queue Q_i for the next level of top-down geometric partitioning and nonlinear scaling, while the remainder is added to Q_g. Whenever Q_i becomes empty, the thread t_i dynamically retrieves clusters from Q_g. The number of clusters N to be retrieved is given by $N = \max(Q_g.size() / NUM\_THREADS, 1)$

Table 1. Comparison of HPWL (×10e6) and runtime (minutes) on ISPD 2005 benchmarks. Each placer ran as a single thread on a 3.2 GHz Intel CPU. FastPlace-DP took 40% of runtime for SimPL and FastPlace.

Benchmark			Capo10.5		NTUPlaces3		FastPlace3.0		mPL6		SimPL	
Name	#Cells	#Nets	HPWL	Time	HPWL	Time	HPWL	Time	HPWL	Time	HPWL	Time
ADAPTEC1	211K	221K	88.14	21.08	81.82	7.62	78.67	2.03	77.93	15.65	*77.42*	*2.01*
ADAPTEC2	255K	266K	100.25	25.44	*88.79*	7.07	94.06	2.88	92.04	16.20	91.01	*2.60*
ADAPTEC3	452K	467K	276.80	62.19	214.83	14.33	214.13	6.51	214.16	48.29	*203.84*	*5.44*
ADAPTEC4	496K	516K	231.30	64.60	195.93	14.55	197.50	6.11	193.89	45.90	*184.70*	*4.88*
BIGBLUE1	278K	284K	110.92	33.42	98.41	12.32	96.65	*3.09*	96.80	20.43	*94.66*	3.88
BIGBLUE2	558K	577K	162.81	64.44	151.55	23.25	155.75	6.11	152.34	54.02	*145.87*	*5.01*
BIGBLUE3	1.10M	1.12M	405.40	146.35	360.66	44.90	365.16	18.87	*344.10*	75.75	351.55	*15.51*
BIGBLUE4	2.18M	2.23M	1016.19	453.72	866.43	100.48	836.20	28.83	829.44	163.15	*790.28*	*23.26*
Geometric mean			*1.19×*	*11.55×*	*1.04×*	*3.32×*	*1.04×*	*1.12×*	*1.03×*	*7.28×*	*1.00×*	*1.00×*

Figure 9. Speedup for conjugate gradient (CG) with SSE instructions, B2B net model construction (B2B), and top-down geometric partitioning and nonlinear scaling (T&N).

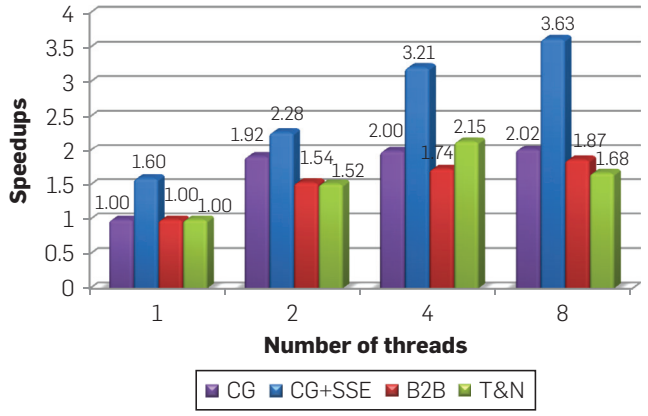
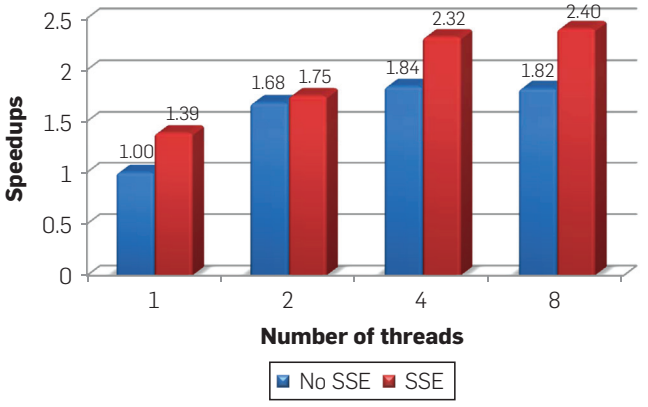


Figure 10. Speedup ratios for global placement on the ISPD 2005 benchmark suite.



Empirical studies evaluated SimPL on an 8-core AMD-based system with four dual-core CPUs and 16GB RAM. Each CPU was Opteron 880 processor running at 2.4 GHz with 1024KB cache. Single-thread execution is compared to eight-thread execution in Figure 9. Our combination of multi-threading and SIMD instruction-level parallelization was 1.6 times faster on average than parallelization based on multi-threading alone. Theoretically, using SIMD instruction-level parallelization may speed up CG by at most four times. However, SIMD-based implementation of SpMxV only provided marginal speedups. This is because irregular memory access patterns of SpMxV prohibit the aligned loading of values (*MOVAPS* or *_mm_load_ps* in Listing 1) to SSE registers. Nevertheless, SSE instructions were helpful elsewhere and contributed to the overall speedup in global placement. The overall speedups in global placement runtimes are shown in Figure 10. Solution quality did not appreciably change, but peak memory usage increased by 1.91 times whereas global placement was 2.4 times faster. The speedups saturate for more than four threads as look-ahead legalization scales poorly. The initial placement stage was accelerated by about three times. While CG remained the runtime bottleneck of SimPL on eight threads (36% of global placement), look-ahead legalization became a close second (>31% of global placement).

7. CONCLUSION

In this work, we developed an algorithm for large-scale VLSI placement. Typical state-of-the-art placers require over 100,000 lines of C++ code, but our self-contained implementation of SimPL uses fewer than 5000 lines.[f] The algorithm is iterative and maintains two placements—one computes a lower bound and one computes an upper bound on the final wirelength. These two placements interact, ensuring stability and fast convergence of the algorithm. The upper-bound placement is produced by a new *feasibility projection* algorithm—*look-ahead legalization*.

The SimPL algorithm has seen rapid adoption since its publication at ICCAD 2010. Two placers[6, 11] based on the SimPL framework finished in top three at the ISPD 2011, DAC 2012, and ICCAD 2012 routability-driven placement contests organized by IBM Research. In particular, He et al.[6] successfully

f The SimPL binary is available upon request.

reimplemented SimPL without having access to our source code. Recent industry and academic software packages for ASIC and FPGA placement are now using similar algorithms. The SimPL algorithm has been extended to multilevel optimization within an industry infrastructure, which currently produces the best HPWL results on average.[10] In this context, SimPL's reduced complexity enables fast implementation, parallel processing, and effective software maintenance. Upper-bound placements help integrate timing and congestion optimizations; the baseline SimPL algorithm has been extended to multi-objective optimization, including power-/thermal-/structure-aware placement as summarized and referenced in Kim and Markov.[9] The recent prosperity of SimPL-derived algorithms suggests the applicability of *look-ahead* techniques to other constrained-optimization problems.

Acknowledgment

This work was supported by Texas Instruments and the Semiconductor Research Corporation (SRC) Task 2264.001 funded by Intel and IBM.

References

1. Alpert, C.J., et al. Techniques for fast physical synthesis. *Proc. IEEE 95*, 3 (2007), 573–599.
2. Brenner, U., Struzyna, M., Vygen, J. BonnPlace: Placement of leading-edge chips by advanced combinatorial algorithms. *IEEE TCAD 27*, 9 (2008), 1607–1620.
3. Chan, T.F., et al. mPL6: Enhanced multilevel mixed-size placement. *ISPD* (2006), 212–214.
4. Chen, T.C., et al. NTUPlace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE TCAD 27*, 7 (2008), 1228–1240.
5. Dagum, L., Menon, R. OpenMP: An industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* (1998), 46–55.
6. He, X., Huang, T., Xiao, L., Tian, H., Cui, G., Young, E.F.Y. Ripple: An effective routability-driven placer by iterative cell movement. *ICCAD* (2011), 74–79.
7. Hu, B., Marek-Sadowska, M. FAR: Fixed-points addition & relaxation based placement. *ISPD* (2005), 161–166.
8. Kahng, A.B., Wang, Q. A faster implementation of APlace. *ISPD* (2006), 218–220.
9. Kim, M.C., Markov, I.L. ComPLx: A competitive primal-dual Lagrange optimization for global placement. *DAC* (2012), 747–752.
10. Kim, M.C., et al. MAPLE: Multilevel adaptive PLacEment for mixed-size designs. *Proc. ISPD* (2012).
11. Kim, M.C., Hu, J., Lee, D., Markov, I.L. A SimPLR method for routability-driven placement. *ICCAD* (2011), 67–73.
12. Kahng, A.B., Lienig, J., Markov, I.L., Hu, J. *VLSI Physical Design: From Graph Partitioning to Timing Closure*, Springer, 2011, 312.
13. Kennings, A.A., Vorwerk, K. Force-directed methods for generic placement. *IEEE TCAD 25*, 10 (2006), 2076–2087.
14. Kleinhans, J.J., et al. GORDIAN: VLSI placement by quadratic programming and slicing optimization. *IEEE TCAD 10*, 3 (1991), 356–365.
15. Nam, G.J., Cong, J. Modern Circuit Placement: Best Practices and Results, Springer, 2007.
16. Pan, M., Viswanathan, N., Chu, C. An efficient & effective detailed placement algorithm. *ICCAD* (2005), 48–55.
17. Raman, S.K., Pentkovski, V., Keshava, J. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro 20*, 4 (2000), 47–57.
18. Roy, J.A., et al. Capo: Robust and scalable open-source min-cut floorplacer. *ISPD* (2005), 224–226.
19. Saad, Y. Iterative methods for sparse linear systems. *SIAM* (2003).
20. Spindler, P., Schlichtmann, U., Johannes, F.M. Kraftwerk2 – A fast force-directed quadratic placement approach using an accurate net model. *IEEE TCAD 27*, 8 (2008), 1398–1411.
21. Trefethen, L.N., Bau, D. Numerical linear algebra. *SIAM* (1997), 296–298.
22. Viswanathan, N., Pan, M., Chu, C. FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. *ASPDAC* (2007), 135–140.
23. Viswanathan, N., et al. RQL: Global placement via relaxed quadratic spreading and linearization. *DAC* (2007), 453–458.

Myung-Chul Kim, Dong-Jin Lee, and Igor L. Markov ([mckima, ejdjsy, markov]@eecs.umich.edu), Department of EECS, University of Michigan, Ann Arbor.