SimPL: An Effective Placement Algorithm

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

Dedicated to the memory of Frank Johannes (1942–2009)

***Abstract*—We propose a self-contained, flat, quadratic global placer that is simpler than existing placers and easier to integrate into timing-closure flows. It maintains lower-bound and upper-bound placements that converge to a final solution. The upper-bound placement is produced by a novel look-ahead legalization algorithm. Our placer SimPL outperforms mPL6, FastPlace3, NTUPlace3, APlace2, and Capo simultaneously in runtime and solution quality, running 7.10 times faster than mPL6 (when using a single thread) and reducing wirelength by 3% on the ISPD 2005 benchmark suite. More significant improvements are achieved on larger benchmarks. The new algorithm is amenable to parallelism, and we report empirical studies with SSE2 instructions and up to eight parallel threads.**

***Index Terms*—Algorithms, layout, multicore, optimization, physical design, placement.**

I. Introduction

GLOBAL placement currently remains at the core of physical design and is a gating factor for downstream optimizations during timing closure [3]. Despite impressive improvements reported by researchers [21] and industry software in the last five years, state-of-the-art algorithms and tools for placement suffer several key shortcomings which are becoming more pronounced at recent technology nodes. These shortcomings fall into four categories: 1) speed; 2) solution quality; 3) simplicity and integration with other optimizations; and 4) support for multithreaded execution. We propose the SimPL algorithm that simultaneously improves results in the first three categories and lends itself naturally to thread-level and instruction-level parallelism on multicore CPUs.

State-of-the-art algorithms for global placement form two families: 1) *force-directed* quadratic placers, such as Kraftwerk2 [27], FastPlace3 [29], and relaxed quadratic spreading and linearization (RQL) [30], and 2) *nonconvex optimization* techniques, such as APlace2 [16], NTUPlace3 [8], and mPL6 [7]. Force-directed quadratic algorithms model total net length by a quadratic function of cell locations and minimize it by solving a large sparse system of linear equations. To discourage cell overlap, *forces* are added pulling cells away from high-density areas. These forces are modeled by *pseudopins* and *pseudonets*, which extend the original quadratic function [14]. They are updated after each linear-system solve until iterations converge. Nonconvex optimization models net length by more sophisticated differentiable functions with linear asymptotic behavior which are then minimized by advanced numerical analysis techniques [16]. Cell density is modeled by functional terms, which are more accurate than forces, but also require updates after each change to placement [8], [16]. Algorithms in both categories are used in the industry or closely resemble those in industry placers.

Manuscript received January 31, 2011; revised June 13, 2011 and August 22, 2011; accepted September 7, 2011. Date of current version December 21, 2011. This paper was recommended by Associate Editor P. Li.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: mckima@eecs.umich.edu; ejdjsy@eecs.umich.edu; imarkov@eecs.umich.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2170567

Tools based on nonconvex optimization achieve the best results reported for academic implementations [8] and electronic design automation (EDA) vendor tools, but are significantly slower, which is problematic for modern flat system-on-chip (SoC) placement instances with tens of millions of movable objects. To scale the basic nonconvex optimization framework, all tools in this family employ *netlist clustering* and *multilevel 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. Due to their complexity, multilevel placers are also harder to maintain, improve, and combine with other physical-design techniques. In particular, clustered netlists complicate accurate static timing analysis, congestion maps, and physical synthesis transformation, such as performance-driven buffering, gate sizing, fanin/fanout optimization, cloning, and others [3].

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. FastPlace3 underperforms mPL6, but the industry tool RQL closely related to FastPlace outperforms state-of-the-art nonconvex placers. Kraftwerk2 is the only competitive *flat* placer (i.e., it does not use clustering) and rivals other force-directed quadratic placers in speed. However, it lags behind in solution quality and poses several challenges, such as quickly solving Poisson's equation, ensuring the convergence of iterations and avoiding halos of unused space around macros. Our experience indicates that the performance of Kraftwerk2 can be uneven, and stability can be achieved with some loss of solution quality [18]. Several placers are described in the book [21] and journal papers [4], [8], [27].

Effective parallelization of computer-aided design optimizations often requires redesign and simplication of entire algorithms to use fewer components, especially standard solvers, to avoid well-known limits to parallelism described by Amdahl's law. On the other hand, recent literature on parallel algorithms and GPGPU programming[1] often focuses on algorithms that are easier to parallelize, but are not the fastest or best-performing available [12], [15], [19]. Such results may be useful to illustrate specific parallelization techniques, but do not justify the need for parallelization. We believe that new EDA tool development should not solely focus on parallel processing, but rather on novel high-performance algorithms amenable to parallel processing.

In this paper, we develop a new, self-contained technique for global placement that ranks as a flat partition-based and force-directed placement algorithm. It maintains lower-bound and upper-bound placements that converge to a final solution. The upper-bound placement is produced by a novel *look-ahead legalization* (LAL) algorithm based on top-down geometric partitioning and nonlinear scaling. Our implementation outperforms published placers simultaneously in solution quality and speed on standard benchmarks. The lower-bound placement is produced by solving a linear system with spreading forces. Our algorithm is simpler, and our attempts to improve overall results using additional modules and extensions from existing placers (such as netlist clustering [7], [16], [29], *iterative local refinement* (ILR) [29], and median-improvement (*BoxPlace*) [18]) were unsuccessful.

In the remainder of this paper, Section II describes the building blocks from which our algorithm was assembled. Section III introduces our key ideas and articulates our solution of the *force modulation* problem. The SimPL algorithm is presented in Section IV along with complexity analysis. Extensions and improvements are discussed in Section V, and empirical validation is described in Section VI. The use of parallelism is discussed in Section VII, and Section VIII summarizes our results.

II. Essential Concepts and Building Blocks

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 rectangular footprint with well-defined area. Some standard cells drive multiple other cells—such interconnects are captured by 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.[2] Some locations of cells may be given initially and fixed. The interconnect objective optimized by global placement is the half-perimeter wirelength (HPWL). For node locations $\vec{x} = \{x_i\}$ and $\vec{y} = \{y_i\}$, $\text{HPWL}_{\mathcal{N}}(\vec{x}, \vec{y}) = \text{HPWL}_{\mathcal{N}}(\vec{x}) + \text{HPWL}_{\mathcal{N}}(\vec{y})$, where

$$\text{HPWL}_{\mathcal{N}}(\vec{x}) = \Sigma_{e \in E}[\max_{i \in e} x_i - \min_{i \in e} x_i]. \tag{1}$$

Efficient optimization algorithms often approximate $\text{HPWL}_{\mathcal{N}}$ by differentiable functions, as illustrated next.

[1]GPGPU programming means general-purpose programming on graphics processing units [12], [15].

[2]In practice, this constraint is enforced for bins of a regular grid. The layout area is subdivided into equal, disjoint, small rectangles, and each rectangle limits total area of cells placed within.

A. 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

$$\Phi_{\mathcal{G}}(\vec{x}, \vec{y}) = \Sigma_{i,j} w_{i,j}[(x_i - x_j)^2 + (y_i - y_j)^2]. \tag{2}$$

Its x and y components are cast in matrix form [4], [27] as follows:

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

The Hessian matrix Q_x captures connections between pairs of movable vertices, while vector $\vec{c}_x$ captures connections between movable and fixed vertices [17, Sec. 4.3.2]. 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 [26]. Since Q_x is symmetric positive definite, CG iterations provably minimize the residual norm. The convergence is monotonic [28], 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*. Our placement algorithm (Section IV-C) deliberately enhances diagonal dominance in Q_x.

B. *Bound2Bound Net Model* [27]

To represent the HPWL objective by the quadratic objective, the netlist $\mathcal{N}$ is transformed in *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 the relative placement of vertices—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:

$$w_{x,ij}^{B2B} = \frac{1}{(p-1)|x_i - x_j|}. \tag{4}$$

For example, 3-pin nets are decomposed into cliques 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 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 [4], [29], [30]. 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 [27].

III. Key Ideas in Our Work

Analytic placement techniques first minimize a function of interconnect length, neglecting overlaps between standard cells, macros, etc. 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 detailed placement).

Our algorithm also starts with pure 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* to coerce the initial locations into a configuration with less overlap, by adding pseudonets to baseline force-directed placement [14].

Each subsequent iteration of our algorithm produces: 1) an almost-legal placement that *overestimates* the final result—through LAL, and 2) an illegal placement that *underestimates* the final result—through linear system solver (LSS). The wirelength gap between lower-bound and upper-bound placements helps monitor convergence (Section IV-C).

A. 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*) [18], [30]. This is unlike previous work, where spreading directions are typically based on *local* information, e.g., placers based on nonconvex optimization use *gradient* information and require a large number of expensive iterations. Kraftwerk2 [27] 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 [18].[3] The authors of RQL [30], 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 paper, the LAL algorithm (Section IV-B), 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. Our placement algorithm does not require exotic components, such as a Poisson-equation solver used by Kraftwerk; our C++ implementation is self-contained.

B. Global Placement with Look-Ahead

The legalized upper-bound placements constructed at every iteration of our placer can be viewed as *look-ahead* because they are used only temporarily and not refined directly. 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

[3]The work in [18] performs force modulation with *line-search* but is not currently competitive with state of the art.

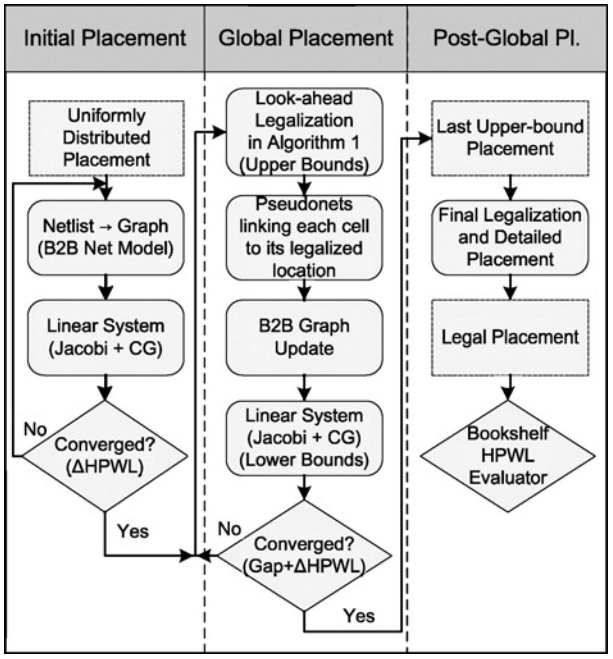
Initial Placement
Global Placement
Post-Global Pl.
Uniformly Distributed Placement
Netlist → Graph (B2B Net Model)
Linear System (Jacobi + CG)
No
Converged? (ΔHPWL)
Yes
Look-ahead Legalization in Algorithm 1 (Upper Bounds)
Pseudonets linking each cell to its legalized location
B2B Graph Update
Linear System (Jacobi + CG) (Lower Bounds)
No
Converged? (Gap+ΔHPWL)
Yes
Last Upper-bound Placement
Final Legalization and Detailed Placement
Legal Placement
Bookshelf HPWL Evaluator

Fig. 1. SimPL algorithm uses placement-dependent B2B net model, which is updated on every iteration. *Gap* refers to the difference between upper and lower bounds.

around fixed obstacles, where local information does not offer sufficient guidance. While not explored in this paper, similar *congestion look-ahead* and *timing look-ahead* based on legalized placements can be used to integrate our placement algorithm into modern timing-closure flows.

IV. Our Global Placement Algorithm

Our placement technique consists of three phases: initial placement (IP), global placement iterations, and postglobal placement (Fig. 1). IP, described next, is mostly an exercise in judicious application of known components. Our main innovation is in the global placement phase. Postglobal placement is straightforward, given current state of the art.

A. Initial Placement

Our initial placement step is conceptually similar to those of other force-directed placers [27], [29], [30]—it entirely ignores cell areas and overlaps, so as to minimize a quadratic approximation of total interconnect length. We found that this step notably impacts the final result. Therefore, unlike FastPlace3 [29] and RQL [30], we use the more accurate B2B net model from [27] reviewed in Section II. 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 iterations (five to seven), regardless of benchmark size, because the relative ordering of locations stabilizes quickly.

B. Look-Ahead Legalization

Consider a set of cell locations with a significant amount of overlap as measured using bins of a regular grid. LAL changes the global positioning of those locations, seeking to remove

Algorithm 1 LAL by top-down geometric partitioning and nonlinear scaling

```
Maximum allowed density $\gamma$, where $0 < \gamma < 1$
Current grid cell size
Floorplan with obstacles
Placement of cells
Queue of bin clusters $Q = \emptyset$

1: Identify $\gamma$-overfilled bins and cluster them // Fig. 2(a)
2: foreach cluster $c$ do
3:   Find a minimal rectangular region $R \supset c$ with density($R$) $\le \gamma$
4.   $R$.level=1
5:   $Q$.enqueue($R$)
6:   while !$Q$.empty() do
7:     $B$=$Q$.dequeue()
8:     if (Area($B$) < 4·grid cell size || $B$.level $\ge$ 10) then
9:       continue
10:    $M$={movable cells in $B$}
11:    if ($B$.level % 2 == 0) then axis_direction $D$=HORIZ
12:    else axis_direction $D$=VERT
13:    $C_c$=$D$-aligned cutline to evenly split cell area in $M$
14:    $C_B$=$D$-aligned cutline to evenly partition whitespace in $B$
15:    ($S_0$, $S_1$)={two subregions of $B$ created by cutline $C_c$}
16:    $M_0$={movable cells in $S_0$}
17:    $M_1$={movable cells in $S_1$}
18:    ($B_0$, $B_1$)={two subregions of $B$ created by cutline $C_B$}
19:    Perform nonlinear scaling on $M_0$ $\perp$ to $D$ in $B_0$
20:    Perform nonlinear scaling on $M_1$ $\perp$ to $D$ in $B_1$
21:    $B_0$.level=$B_1$.level=$B$.level+1
22:    $Q$.enqueue($B_0$)
23:    $Q$.enqueue($B_1$)
24:  end while
25: end foreach
```

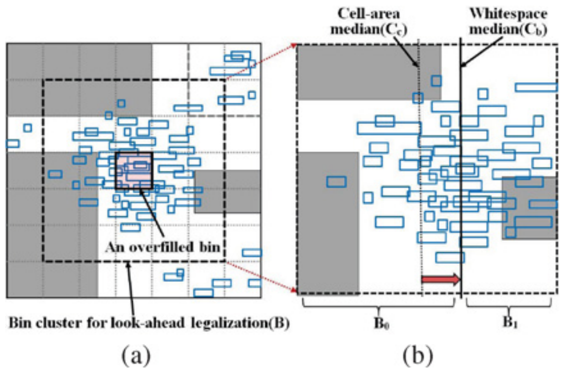

Fig. 2. (a) Clustering of overfilled bins in Algorithm 1 and (b) adjustment of cell-area to whitespace median by nonlinear scaling (also see Fig. 3). Movable cells are shown in blue, obstacles in solid gray.

most of the overlap (with respect to the grid) while preserving the relative ordering. This task can be formulated at different geometric scales by varying the grid. The quality of LAL is measured by its impact on the entire placement flow. Our LAL is based on top-down recursive geometric partitioning and nonlinear scaling, as outlined in Algorithm 1. Cutlines C_c and C_B are chosen to be vertical at the top level and they alternate between horizontal and vertical directions with each successive level of top-down geometric partitioning.

1) *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, and LAL is performed on such clusters. For each cluster, we find a minimal containing rectangular region with density $\le \gamma$ (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.[4] If relative placement must be preserved, overlap can be reduced by means of x-sorting 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 subregions (Fig. 2).

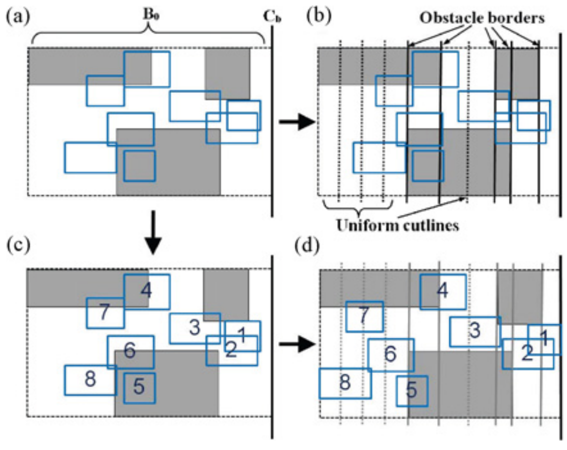

Fig. 3. (a) Nonlinear scaling in a region with obstacles. (b) Formation of C_B-aligned stripes. (c) Cell sorting by distance from C_B. (d) Greedy cell positioning.

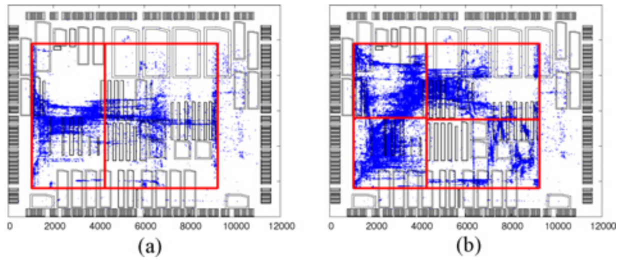

Fig. 4. (a) Nonlinear scaling after the first vertical cut and (b) two subsequent horizontal cuts (adaptec1) from intermediate steps between iterations 0 and 1 in Fig. 7.

Nonlinear scaling in one direction is illustrated in Fig. 3, 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 (by up to ten evenly distributed cutlines) if its available area exceeds 1/10 of the region's available area. Movable cells in the corresponding subregion 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 stripe that is not filled yet.

For each stripe, we calculate the available site area A_a and consider the stripe filled when the area of assigned cells reaches γA_a. Cell locations within each stripe are linearly

[4]In the presence of whitespace, the placer can move cells around without changing their relative ordering [2]. Removing whitespace suppresses this degree of freedom, giving fewer choices to the placer.

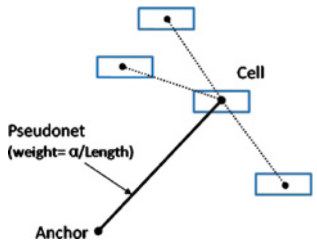

Fig. 5. Anchor with a pseudonet.

scaled from current locations (nonlinearity arises from different scaling in different stripes).

LAL applies nonlinear scaling in alternating directions, as illustrated in Fig. 4 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, LAL (Algorithm 1) considers each subregion individually and selects different horizontal cutlines. Four rounds of nonlinear scaling follow, spreading cells over the region's expanse (Fig. 4).

Despite a superficial similarity to cell-shifting in FastPlace [29], our nonlinear scaling does not use cell locations to define bins/ranges, or map ranges onto a uniform grid.

2) *Cutline Shifting:* Median-based cutlines are neither necessary nor sufficient for good solution quality. We therefore adopt a fast cutline positioning technique from [24]. On benchmarks whose obstacles cover <20% of total sites area, we find cutline positions C_c minimizing net cut for the top two levels of top-down geometric partitioning, with <60% of cell area per partition. We record the ratio ρ of cell areas in the two partitions and adjust the region's C_B cutline to the position that partitions the region's available area with the same ratio ρ. A related technique called analytic constraint generation was developed at IBM in the context of min-cut placement, and their paper [2] describes relevant intuition.

C. Global Placement Iterations

1) *Using Legalized Locations as Anchors:* Solving an unconstrained linear system results in a placement with significant amount of overlap. To pull cells away from their initial positions, we gradually perturb the linear system. As explained in Section IV-B, at each iteration of our global placement, top-down geometric partitioning and nonlinear scaling generates 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 II, we note that connections to fixed locations do not increase the size of the Hessian matrix Q, and only contribute to its diagonal elements [17, Sec. 4.3.2]. 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 $\alpha = 0.01 \cdot (1 + \text{iterationNumber})$. At early iterations, small α values weaken spreading forces, giving greater significance to interconnect and more freedom to the LSS. 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.

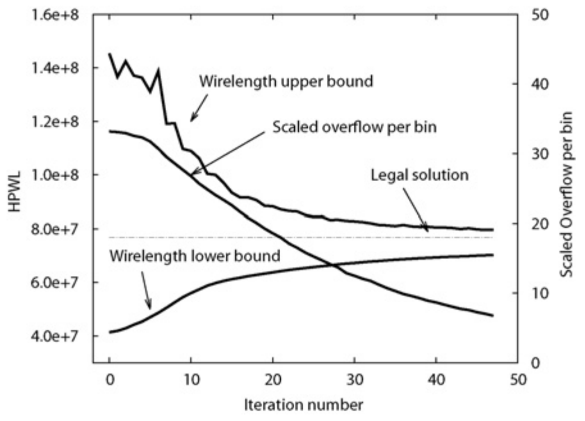

Fig. 6. Lower and upper bounds for HPWL, the scaled overflow per bin of the lower-bound placement at each iteration, and HPWL of the legal placement (adaptec1).

2) *Grid Resizing:* To identify γ-overfilled bins, we overlay a uniform grid over the entire layout. The grid size is initially set to $S_{\text{init}} = 100 \times 100$ to accelerate the LAL. However, in order to accurately capture the amount of overlap, the grid cell size decreases by $\beta = 1.06$ at each iteration of global placement until it reaches 4× the average movable cell size.[5] Grid resizing also affects the clustering of γ-overfilled bins during LAL (Section IV-B), effectively limiting the amount of cell movement and encouraging convergence at later iterations. A progression of global placement is annotated with HPWL values in Fig. 7. The upper-bound placements on the right appear blocky in the first iteration, but gradually refine with grid resizing.

3) *Convergence Criteria:* A convergence criterion similar to that in Section IV-A can be adopted in global placement. We alternate: a) LAL; b) updates to anchors and the B2B net model; and c) solution of the linear system, until HPWL of solutions generated by LAL stops improving. Unlike in the IP step, however, HPWL values of upper-bound solutions oscillate during the first four to seven iterations, as illustrated in Fig. 6. To prevent premature convergence, we monitor the gap between the lower and upper bounds. Global placement continues until: a) the gap is reduced to 25% of the gap in the tenth iteration and upper-bound solution stops improving, or b) the gap is smaller than 10% of the gap in the tenth iteration. On the ISPD 2005 benchmark suite, this convergence criterion entails 26–47 iterations of global placement. The final set of locations (global placement) is produced by the last LAL as shown in Fig. 1.

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.

[5]This is similar to mesh sizing for finite-element methods in numerical analysis and especially in adaptive mesh refinement. Parameters can be optimized for benchmark suites using binary search. However, we have not tuned parameters to individual benchmarks.

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 LAL. In the extreme, very high pseudonet weights force the lower-bound placement so close to the upper-bound placement, that LAL does not change it, resulting in immediate convergence.[6] In practice, pseudonet weights are changed gradually to ensure low interconnect length. After the first few iterations, one typically observes monotonic convergence, as illustrated in Fig. 6.

D. Asymptotic Complexity Analysis

Modern placement algorithms are too complicated for asymptotic complexity analysis, but the bottlenecks of the SimPL algorithm yield to traditional analysis.

The runtime of global placement iterations is dominated by the CG solver and LAL. The complexity of each CG invocation is $O(m\sqrt{\kappa})$, where κ is the conditioning number of the matrix and m is the number of nonzero elements [28]. 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.[7] Since $m \leq (d + 1)n$ for n cells,[8] CG runs in $O(n\sqrt{\kappa})$ time.

Asymptotic runtime of LAL 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 LAL 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, LAL runs in $O(n \log^2 n)$ time.

We have observed that due 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.

Empirically, SimPL requires <50 placement iterations, even for circuits with millions of cells. While the number of iterations might grow for larger circuits, this growth is very slow—possibly a polylog function of n. Empirical results in Section VI show that SimPL's advantage in runtime and solution quality over its closest competitor (FastPlace3) increases on larger netlists. Min-cut placement (Capo) exhibits asymptotic complexity $O(n \log^2 n)$, but lags behind SimPL in runtime and quality.

[6]This convergence argument only assumes that LAL does not change an upper-bound placement. It does not make any other assumptions about the LAL algorithm or consistency of its results between iterations. Neither does it say anything about the quality of results.

[7]Even with large macros, whose number is limited by design area.

[8]Including diagonal matrix elements.

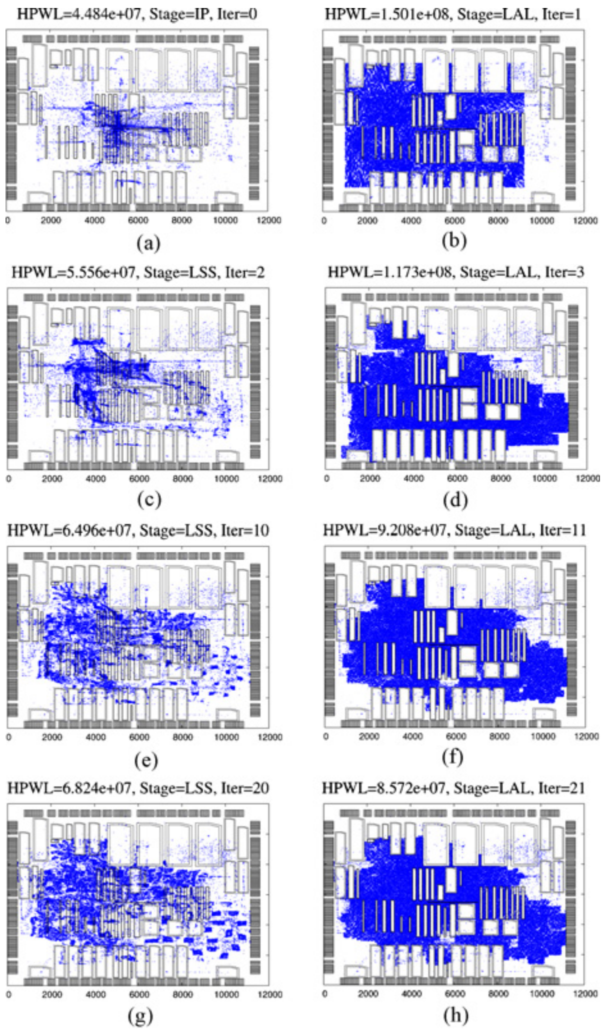

Fig. 7. Progression of global placement snapshots from different iterations and algorithm steps (adaptec1). (a) HPWL= 4.484e+07, Stage=IP, Iter=0. (b) HPWL= 1.501e+08, Stage=LAL, Iter=1. (c) HPWL= 5.556e+07, Stage=LSS, Iter=2. (d) HPWL= 1.173e+08, Stage=LAL, Iter=3. (e) HPWL= 6.496e+07, Stage=LSS, Iter=10. (f) HPWL= 9.208e+07, Stage=LAL, Iter=11. (g) HPWL= 6.824e+07, Stage=LSS, Iter=20. (h) HPWL= 8.572e+07, Stage=LAL, Iter=21. Left-side placements show lower bounds and right-side placements show upper bounds.

Space complexity of our algorithms is linear in the size of the input, and our implementations require a modest amount of memory.

V. Extensions and Improvements

The algorithm in Section IV can be improved in terms of runtime and solution quality. However, some of our attempts at improvement were unsuccessful. We report them here to warn the reader about their futility.

A. Selecting Windows for LAL

During early global iterations, most movable cells of the lower-bound placement reside near the center of the layout region (Fig. 7). In such cases, there is usually one expanded minimal rectangular region (cluster) that will encompass most of γ-overfilled bins. However, as global iterations progress, γ-overfilled bins will be scattered around the layout region, and multiple clusters of bins may exist. In our implementation, we process γ-overfilled bins in the decreasing order of density. Each expansion stops when the cluster's density drops to γ or the cluster abuts the boundaries of previously processed clusters. This strategy may generate incompletely expanded clusters, especially in midstages of global placement iterations. However, as the densest bins are processed first, the number of regions with peak density is guaranteed to decrease at every iteration except when the peak density itself decreases. At each iteration of global placement, LAL is repeated up to ten times with increasing grid cell sizes until maximal density is decreased below γ.

B. Improving Asymptotic Runtime Complexity of LAL

As explained in Section IV-D, asymptotic runtime of LAL is largely determined by sorting cells by positions in directions perpendicular to cutlines. This sorting occurs at each level of top-down geometric partitioning. One way to improve asymptotic runtime complexity of LAL is to invoke sorting less often, given that LAL is to preserve the relative ordering among cells. Instead of sorting cells in each subregion, we first establish two cell arrays sorted by x-coordinates and y-coordinates, respectively. At the second level of top-down geometric partitioning, two subregions inherit corresponding cells in-order from two sorted arrays of cells. In this way, if sorting is performed once at the top level of geometric partitioning, sorting at all successive levels can be replaced by selecting appropriate cells that belong to current region in-order from two sorted arrays of higher level. This improves asymptotic runtime complexity of LAL from $O(n \log^2 n)$ time to $O(n \log n)$.

C. Unsuccessful Attempts at Improvement

Compared to other placement algorithms, SimPL uses a very modest set of interconnect optimizations. Therefore, we experimented with adding to SimPL several algorithms that were reported essential to the performance of other placers.

1) Our first attempt was to use netlist clustering to extend SimPL into a multilevel algorithm [7], [16], [29]. To this end, we implemented BestChoice clustering [20] used in FastPlace3 and were able to match its performance observed in FastPlace3 logs. This accelerated the initial CG solve in SimPL by about 2×, with essentially the same quality of results, but unclustering increased the amount of cell overlaps, and the refinement techniques that we tried were either ineffective or too time-consuming.
2) In our second attempt, we implemented ILR [29], which is a stage of FastPlace-global where it spends 40–50% of its runtime. ILR is a simple move-based algorithm that postprocesses results of quadratic placement by relocating cells to nearby grid bins, while keeping track of both HPWL and cell density. ILR did improve the results of our early prototypes, but adding it to SimPL does not improve final results. We believe that our LAL algorithm provides sufficient density control with a moderate increase in HPWL. We also tried, unsuccessfully, the median-improvement (BoxPlace) algorithm from [18], which moves single cells to their HPWL-optimal locations, while considering adjacent cells fixed.
3) In a third attempt, we evaluated *ad hoc* force modulation used in RQL [30] that neglects 10% strongest forces. Sweeping the range from 1% to 10% did not reveal any improvement in our experiments.
4) In our fourth attempt at improvement, we reordered vertices in the netlist to improve memory locality for each invocation of CG. This technique is often applied to the matrices of linear systems and is known to reduce cache misses and runtime. We implemented the reverse Cuthill–McKee reordering, which is standard in numerical analysis. The locality of nets has significantly improved. However, CG did not run faster on any of our benchmarks—the default ordering in our benchmarks was already good enough.
5) In summary, we obtain state-of-the-art results without extensions reported essential to other placers (FastPlace3 [29], force-directed placement [18], and RQL [30]). We have also experimented with several preconditioners for CG, but found the simplest of them—the diagonal (Jacobi) preconditioner—to work best in our application.

VI. Empirical Validation

Our implementation was written in C++ and compiled with g++ 4.4.0. Unless indicated otherwise, benchmark runs were performed on an Intel Core i7 Quad CPU Q660 Linux workstation running at 3.2 GHz, using only one CPU core. We compared SimPL to other academic placers on the ISPD 2005 placement contest benchmark suite with target density γ=1.0. Focusing on global placement, we delegate final legalization (into rows and sites) and detailed placement to FastPlace-DP [22], but postprocess it by a greedy cell-flipping algorithm from Capo [6]. HPWL of solutions produced by each placer is computed by the GSRC Bookshelf Evaluator [1].

A. Analysis of Our Implementation

The SimPL global placer is a stand-alone tool that includes I/O, IP, 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. There are four command-line parameters that affect performance—two for grid resizing (initial and step), and two for pseudonet weighting (initial and step). In all experiments, we used default values described in Section IV.

Running in a single thread, SimPL completes the entire ISPD 2005 benchmark suite in 1 h 18 min, placing the largest benchmark, bigblue4 (2.18M cells), in 38 min using 2.1 GB of memory. We report the runtime breakdown on bigblue4 according to Fig. 1, excluding 1.4% runtime for I/O. IP 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. Inserting pseudonets takes 0.8%, and LAL 17.7%. Postglobal placement takes 46.2%, predominantly

TABLE I

LEGAL HPWL ($\times$ 10E6) AND TOTAL RUNTIME (MINUTES) COMPARISON ON THE ISPD 2005 BENCHMARK SUITE

Benchmark	APlace2.0		Capo10.5		FastPlace3.0		mPL6		NTUPlace3		SimPL	
Size (#Cells)	HPWL	Time	HPWL	Time	HPWL	Time	HPWL	Time	HPWL	Time	HPWL	Time
AD1 211K	78.35	35.02	88.14	25.95	78.16	2.50	77.93	18.36	81.82	8.20	**76.87**	**2.47**
AD2 255K	95.70	50.57	100.25	36.06	93.56	3.66	92.04	19.91	**88.79**	7.57	90.37	**3.40**
AD3 452K	218.52	119.53	276.80	78.19	213.85	8.48	214.16	58.92	214.83	15.62	**206.38**	**6.68**
AD4 496K	209.28	131.57	231.30	79.32	198.17	7.10	193.89	55.95	195.93	16.18	**186.00**	**5.88**
BB1 278K	100.02	44.91	110.92	41.78	96.32	3.77	96.80	22.82	98.41	13.22	**95.85**	**3.47**
BB2 558K	153.75	100.96	162.81	80.55	154.91	9.62	152.34	61.55	151.55	26.17	**143.56**	**7.58**
BB3 1.10M	411.59	209.24	405.40	182.94	365.59	21.59	344.10	85.23	360.07	51.08	**336.19**	**13.02**
BB4 2.18M	871.29	489.05	1016.19	567.15	834.19	40.93	829.44	189.83	866.43	115.06	**796.78**	**37.37**
Geomean	**1.09×**	**15.34×**	**1.20×**	**12.17×**	**1.05×**	**1.20×**	**1.03×**	**7.10×**	**1.05×**	**3.05×**	**1.00×**	**1.00×**

Each placer ran as a single thread on a 3.2 GHz Linux workstation. HPWL was computed by the GSRC Bookshelf Evaluator [1]. Full names of benchmarks are abbreviated: "ad" for "adaptec" and "bb" for "bigblue."

in detailed placement. Greedy orientation improvement and HPWL evaluation were almost instantaneous.

B. Comparisons to State-of-the-Art Placers

We compared SimPL to other placers whose binaries are available to us. We run each available placer,[9] including SimPL, in *default* mode and show results in Table I. The HPWL results reported by APlace2 [16], Capo10.5 [6], [25], and mPL6 [7] were confirmed by the GSRC Bookshelf Evaluator. However, FastPlace3 [29] reported lower HPWL by 0.25–0.96%. For consistency, we report the readings of the GSRC Bookshelf Evaluator.

SimPL found placements with the lowest HPWL for seven out of eight circuits in the ISPD 2005 benchmark suite (no parameter tuning to specific benchmarks was employed). On average, SimPL obtains wirelength improvement of 7.73%, 16.47%, 4.38%, 2.98%, and 4.48% versus APlace2, Capo10.5, FastPlace3, mPL6, and NTUPlace3, respectively. SimPL was also the fastest among the placers on all eight circuits. It is 7.01 times faster than mPL6, which appears to be the strongest pre-existing placer. SimPL is 1.20 times faster than FastPlace3, which has been the fastest academic placer so far.

While we managed to obtain almost all best-performing academic placers in binaries, RQL reportedly outperforms mPL6 in HPWL by a small amount [30]. Comparing our HWPL results to numbers in [30], we observe five wins for SimPL and three losses. RQL is 3.1 times faster than mPL6, making it more than twice as slow as SimPL.

C. Scalability Study

To demonstrate SimPL's scalability to larger netlists, we generated variants of ISPD 2005 benchmarks with netlists that are twice as big with the same area utilization. In such a double-sized benchmark, each movable cell is split in two cells of smaller size, and each connection to the original cell is inherited by one of the split cells. Additionally, the two split cells are connected by a new two-pin net (Fig. 8).

We compared SimPL to FastPlace3, mPL6, and NTUPlace3 on the double-sized benchmark suite and show results in Table II. mPL6 could not finish bigblue4. For bigblue3, FastPlace-DP was unable to completely legalize solutions produced by FastPlace3-global, hence we postprocessed FastPlace-DP with Capo10.5's legalizer.

[9]The KraftWerk2 binary we obtained did not run on our system.

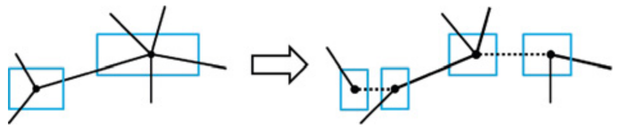

Fig. 8. Generation of double-sized netlists.

SimPL was the fastest among the placers on all eight circuits. It is 8.96 times faster than mPL6, and 1.49 times faster than FastPlace3. SimPL also found placements with the lowest HPWL for six out of eight circuits in the double-sized ISPD 2005 benchmark suite (no parameter tuning to specific benchmarks was employed). Comparing results in Table II to those in Table I, we observe that our placer has greater advantage on larger benchmarks. Furthermore, our runtime comparisons include detailed placement, but if SimPL is compared to FastPlace3-global without detailed placement, the average speedup increases to 1.82 times from 1.58 times.

Compared to other placers, our implementation uses a modest amount of memory—1.65 times and 2.39 times less than mPL6 and NTUPlace3, respectively, and 1.61 times more than FastPlace3. SimPL is using more memory than FastPlace3 when it constructs sparse matrices based on the B2B net model.

VII. SPEEDING UP PLACEMENT USING PARALLELISM

Further speedup is possible for SimPL on workstations with multicore CPUs.

A. Algorithmic Details

Runtime bottlenecks in the sequential variant of the SimPL algorithm (Section VI-A)—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* [13] scheme to the Hessian Matrix Q, where different blocks of rows are assigned to different threads using OpenMP [11]. A critical kernel operation in CG is the sparse matrix-vector multiply (SpMxV). Memory bandwidth is a known performance bottleneck in a uniprocessor environment [10], and its impact is likely to aggravate when multiple cores access the main memory through a common bus. We reduce memory

TABLE II

Legal HPWL ($\times$10e6), Total Runtime (Minutes), and Peak Memory Usage (Gigabytes) Comparison on the Double-Sized ISPD 2005 Benchmark Suite

Ckts	FastPlace3.0			mPL6			NTUPlace3			SimPL		
	HPWL	Time	Memory	HPWL	Time	Memory	HPWL	Time	Memory	HPWL	Time	Memory
Ad1x2	80.30	3.67	**0.20**	79.11	24.95	0.43	80.20	19.6	0.72	**77.02**	**3.05**	0.36
Ad2x2	98.88	6.06	**0.23**	93.64	38.00	0.82	**91.56**	17.6	0.86	92.41	**4.11**	0.36
Ad3x2	258.71	13.47	**0.42**	232.87	88.71	1.11	225.32	44.7	1.62	**215.56**	**7.58**	0.71
Ad4x2	219.35	11.54	**0.44**	206.24	85.72	1.16	197.90	39.6	1.76	**193.18**	**7.25**	0.69
Bb1x2	97.93	5.68	**0.25**	100.37	30.35	0.53	99.33	22.0	0.93	**96.39**	**4.71**	0.43
Bb2x2	164.74	12.13	**0.49**	159.24	79.84	1.22	154.47	44.2	1.94	**148.43**	**9.02**	0.77
Bb3x2	515.61	49.89	**0.93**	395.26	172.96	3.38	**386.65**	154.9	3.82	403.40	**22.24**	1.44
Bb4x2	865.30	56.36	**1.94**	fail	fail	fail	866.78	267.9	7.86	**854.64**	**42.43**	2.91
Geomean	**1.10**×	**1.49**×	**0.62**×	**1.04**×	**8.96**×	**1.65**×	**1.02**×	**4.40**×	**2.39**×	**1.00**×	**1.00**×	**1.00**×

The failure of mPL6 in bigblue4×2 does not appear to be caused by out-of-memory conditions.

TABLE III

Speedup Ratios for CG, B2B Net Model Construction, and Top-Down Geometric Partitioning and Nonlinear Scaling (T&N) on the ISPD 2005 Benchmark Suite

Ckts	One Core	Two Threads				Four Threads				Eight Threads			
	CG + SSE	CG	CG + SSE	B2B	T&N	CG	CG + SSE	B2B	T&N	CG	CG + SSE	B2B	T&N
AD1	1.37	1.88	2.21	1.45	1.41	2.03	2.87	1.64	1.59	1.92	3.28	1.91	1.48
AD2	1.61	1.77	2.09	1.50	1.53	2.12	3.01	2.04	2.17	2.06	3.22	1.98	1.40
AD3	1.57	1.76	2.20	1.48	1.62	1.88	3.17	1.62	2.25	2.00	3.55	1.79	1.66
AD4	1.50	1.65	2.07	1.51	1.56	1.81	3.03	1.58	2.17	1.77	3.33	1.72	1.36
BB1	1.57	2.03	2.11	1.27	1.71	2.02	3.14	1.66	2.93	2.05	3.70	1.78	2.93
BB2	1.62	2.07	2.24	1.48	1.49	1.72	2.97	1.68	1.89	1.79	3.50	1.74	1.58
BB3	1.54	1.53	2.25	1.60	1.32	1.68	3.04	1.64	2.04	1.81	3.30	1.85	1.32
BB4	2.01	2.63	3.04	2.01	1.59	2.71	4.48	2.02	2.12	2.76	5.12	2.18	1.68
GM	**1.59**×	**1.89**×	**2.26**×	**1.53**×	**1.52**×	**1.98**×	**3.18**×	**1.73**×	**2.12**×	**2.00**×	**3.59**×	**1.86**×	**1.62**×

Runtimes are compared to single-threaded execution without support of SSE instructions.

```
// inner product of two float vectors x and y
float inner_product(vector<float>&x, vector<float>&y)
{
  float p_acc[4], inner_product=(float)0.;
  __m128 X, Y, acc = _mm_set_zero_ps();
  unsigned i;
  #pragma omp parallel for schedule(static)
  private(X,Y) lastprivate(i) reduction(+:acc)
  num_threads(NUM_CORES)
  for (i=0 ; i<=x.size()-4 ; i+=4)
  {
     X = _mm_load_ps(&x[i]);
     Y = _mm_load_ps(&y[i]);
     acc = _mm_add_ps(acc, _mm_mul_ps(X, Y));
  }
  _mm_store_ps(p_acc, acc);
  inner_product = p_acc[0]+p_acc[1]+p_acc[2]+p_acc[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.

bandwidth demand of SpMxV by using the compressed sparse row [26] memory layout for the Hessian matrix Q.

In addition to thread-level parallelism, our implementation makes use of streaming single instruction, multiple data (SIMD) extensions level 2 (SSE2) [23] (through g++ intrinsics) that perform several floating-point operations at once. SSE2 instructions are extensively used in our CG solver. Since SSE2 instructions are available in most modern CPUs, we used them in the default mode evaluated in Tables I and II. The overall speedup due to parallelism varies between different hardware systems, as it depends on the relation between CPU speed and memory bandwidth.

After we parallelized the main bottlenecks, we noticed that LAL started consuming a significant fraction of overall runtime. Fortunately, top-down geometric partitioning and nonlinear scaling are amenable to parallelization as well. Notably, top-down partitioning generates an increasing number of subtasks of similar sizes which can be solved in parallel. Let Q be the global queue of bin clusters, as defined in Algorithm 1, and each thread has a private queue of bin clusters Q_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 on such bin cluster, each thread generates two subclusters with similar numbers of cells. Then the thread t_i adds only one of two subclusters to its own cluster queue Q_i for the next level of top-down geometric partitioning and nonlinear scaling, while the remainder is added to the global cluster queue Q. Whenever Q_i of a thread t_i becomes empty, the thread t_i dynamically retrieves clusters from the global cluster queue Q. The number of clusters to be retrieved N is given by

$$N = \max\left(Q.size()/N_{\text{threads}}, 1\right)$$

where

N_{threads} is the total number of threads.

B. Empirical Studies

As part of our empirical validation, we ran SimPL on an 8-core AMD-based system with four dual-core CPUs and 16 GB RAM. Each CPU was Opteron 880 processor running at 2.4 GHz with 1024 KB cache. Single-thread execution was compared to eight-thread execution as shown in Table III. Our combination of multithreading and SIMD instruction-level parallelization was 1.6 times faster on average than parallelization based on multi-threading alone. Theoretically,

TABLE IV
SPEEDUP RATIOS FOR GLOBAL PLACEMENT ON THE ISPD 2005 BENCHMARK SUITE

Ckts	Two Threads		Four Threads		Eight Threads	
	No SSE	SSE	No SSE	SSE	No SSE	SSE
AD1	1.70	1.71	1.76	2.03	1.71	2.23
AD2	1.75	1.73	1.91	2.43	1.90	2.35
AD3	1.59	1.72	1.79	2.30	1.81	2.40
AD4	1.55	1.65	1.75	2.24	1.67	2.26
BB1	1.75	1.67	2.17	2.56	2.18	2.67
BB2	1.70	1.72	1.67	2.22	1.66	2.37
BB3	1.49	1.75	1.71	2.28	1.65	2.28
BB4	1.94	2.12	2.01	2.55	2.03	2.69
GM	**1.68×**	**1.75×**	**1.84×**	**2.32×**	**1.82×**	**2.40×**

Runtimes are compared to single-threaded execution without support of SSE instructions.

using SIMD instruction-level parallelization may speedup CG by at most four times. However, SIMD-based implementation of SpMxV only provided marginal speedups and was not worth the development effort. 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 in other parts of the code and contributed to the overall speedup in global placement, as illustrated in Table IV.

We note that LAL operates on large datasets, but performs little computation per datum, which limits its performance by memory bandwidth. The amount of work per thread is so small that the overhead of thread creation outweighed the benefits. As a result, this part of SimPL scales poorly to >4 threads on available hardware, although this is probably not a fundamental limitation of the algorithm.

The overall speedups in global placement runtimes are shown in Table IV. Solution quality did not appreciably change, but peak memory usage increased by 1.91 times whereas runtime of global placement iterations was reduced by 2.4 times on average. The speedups saturate for more than four threads as LAL scales poorly. The IP stage was accelerated by about three times. While CG remained the runtime bottle neck of SimPL on eight threads (36% of global placement), LAL became a close second (>31% of global placement).

VIII. Conclusion and Future Work

In this paper, we developed a new, flat, partition-based, and force-directed quadratic global placer. Unlike other state-of-the-art placers, it was rather simple, and our self-contained implementation included fewer than 5000 lines of C++ code. The algorithm was iterative and maintained two placements—one computed a lower bound and one computed an upper bound on final wirelength. These two placements interact, ensuring stability and fast convergence of the algorithm. The upper-bound placement was produced by a new *look-ahead legalization* algorithm, based on top-down geometric partitioning and nonlinear scaling, and converged to final cell locations. In contrast, all analytic algorithms we reviewed (both force-directed quadratic and nonconvex) derived their final solution from a lower-bound placement.

The use of partition-based techniques in upper-bound placements offers a solution to the force-modulation problem [18], [30] and removes the need for the so-called *hold forces* used by several force-directed placers.[10] As discussed in Section III, upper-bound placements perform an *area look-ahead*[11] that is instrumental in the handling of layout obstacles. APlace2, NTUPlace3, mPL6, as well as some force-directed placers, model obstacles by additional smoothened penalty terms in the objective function. Not only such terms introduce extra work, but they also add imprecisions to modeling. For similar reasons, SimPL avoids netlist clustering used by other placers. We have implemented several other techniques essential to well-known placers, such as BoxPlace [18], ILR [29], and *ad hoc* force modulation [30], but they did not improve SimPL results.

SimPL is highly competitive on ISPD 2005 benchmarks where it outperforms every placer available to us in binary both by solution quality and runtime. SimPL's advantage in runtime and solution quality over FastPlace3 and mPL6 grows on larger netlists. However, its most compelling advantages over prior state-of-the-art deal with practical uses of placement in modern timing-closure design flows: 1) the reduced complexity of SimPL allows for fast implementation, parallel processing, and effective software maintenance, and 2) the upper-bound placements facilitate tighter integration of timing and congestion optimizations into the global placement process, improving the speed and quality of physical synthesis.

The SimPL algorithm saw rapid adoption since its first publication at ICCAD 2010. At the ISPD 2011 placement contest, the winning team successfully implemented SimPL without having access to our source code. To our knowledge, at least two major EDA vendors are now using similar placement algorithms, and our own work with a state-of-the-art industry placer quickly produced significant improvements that will be discussed in future publications.

The implementation of SimPL described in this paper is designed for standard-cell layouts and does not yet handle movable macroblocks. Our recent industry experience suggests that the majority of modern large-scale placement instances in practice do not require this feature, as their macroblocks are fixed. However, mixed-size placement is useful for some mixed-signal SoCs, and we are addressing it in our ongoing work.[12]

Attempting to explain theoretically the strong performance of our placement algorithm, we have noticed similarities to *primal-dual* algorithms for convex [31] and combinatorial [5] optimization. Primal-dual methods maintain lower and upper bounds, expressed by primal and dual solutions that eventually converge to an optimal feasible solution. The interpretation of duality as swapping the problem's constraints for the objective function [31] is also consistent with our algorithm—LAL corresponds to imposing a no-overlap constraint while relaxing

[10]Hold forces are used to ensure that the current placement is a force equilibrium. Then move forces are added so that the placement can be improved. While this technique is needed to ensure convergence of iterations, SimPL relies on anchors and pseudonets to ensure convergence.

[11]The concept of *area look-ahead* was proposed in [9] for block-packing by nested bisection, where it checks if a given bisection admits a legal block packing in each partition. Area look-ahead was not used in [9] to spread standard cells from dense regions.

[12]Academic placers typically introduce this feature in dedicated publications, rather than in the first publication describing the baseline algorithm.

the linear constraints that capture the global minimum of the quadratic wirelength objective. The key to the success of primal-dual algorithms [5], [31] is the observation that alternating progress in *primal* and *dual* solutions, i.e., improving the cost of feasible solutions and tightening the constraints for low-cost solutions, typically leads to faster convergence compared to one-sided optimizations. This effect is empirically observed in Section VI where SimPL is compared to pre-existing placement algorithms, all of which are one-sided.

References

[1] S. N. Adya and I. L. Markov. (2005). *Executable Placement Utilities* [Online]. Available: http://vlsicad.eecs.umich.edu/BK/PlaceUtils

[2] C. J. Alpert, G.-J. Nam, and P. G. Villarrubia, "Effective free space management for cut-based placement via analytical constraint generation," *IEEE Trans. Comput.-Aided Des.*, vol. 22, no. 10, pp. 1343–1353, Oct. 2003.

[3] C. J. Alpert, S. K. Karandikar, Z. Li, G.-J. Nam, S. T. Quay, H. Ren, C. N. Sze, P. G. Villarrubia, and M. C. Yildiz, "Techniques for fast physical synthesis," *Proc. IEEE*, vol. 95, no. 3, pp. 573–599, Mar. 2007.

[4] U. Brenner, M. Struzyna, and J. Vygen, "BonnPlace: Placement of leading-edge chips by advanced combinatorial algorithms," *IEEE Trans. Comput.-Aided Des.*, vol. 27, no. 9, pp. 1607–1620, Sep. 2008.

[5] N. Buchbinder and J. Naor, *The Design of Competitive Online Algorithms Via a Primal-Dual Approach*. Hanover, MA: NOW Publishers, 2009.

[6] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Can recursive bisection alone produce routable placements?" in *Proc. DAC*, 2000, pp. 477–482.

[7] T. F. Chan, J. Cong, J. R. Shinnerl, K. Sze, and M. Xie, "mPL6: Enhanced multilevel mixed-size placement," in *Proc. ISPD*, 2006, pp. 212–214.

[8] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUPlace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Trans. Comput.-Aided Des.*, vol. 27, no. 7, pp. 1228–1240, Jul. 2008.

[9] J. Cong, M. Romesis, and J. R. Shinnerl, "Fast floorplanning by look-ahead enabled recursive bipartitioning," *IEEE Trans. Comput.-Aided Des.*, vol. 25, no. 9, pp. 1719–1732, Sep. 2006.

[10] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Proc. Euromicro Int. Conf. PDP*, Feb. 2008, pp. 283–292.

[11] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computat. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.

[12] M. Garland, "Sparse matrix computations on manycore GPU's," in *Proc. DAC*, 2010, pp. 2–6.

[13] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell, "Exploring the cache design space for large scale CMPs," *ACM SIGARCH Comput. Architect. News*, vol. 33, no. 4, pp. 24–33, Nov. 2005.

[14] B. Hu and M. Marek-Sadowska, "FAR: Fixed-points addition and relaxation based placement," in *Proc. ISPD*, 2005, pp. 161–166.

[15] D. A. Jamsek, "Designing and optimizing compute kernels on NVIDIA GPUs," in *Proc. ASPDAC*, 2009, pp. 224–229.

[16] A. B. Kahng and Q. Wang, "A faster implementation of APlace," in *Proc. ISPD*, 2006, pp. 218–220.

[17] A. B. Kahng, J. Lienig, I. L. Markov, and J. Hu, *VLSI Physical Design: From Graph Partitioning to Timing Closure*. New York: Springer, 2011, pp. 1–312.

[18] A. A. Kennings and K. Vorwerk, "Force-directed methods for generic placement," *IEEE Trans. Comput.-Aided Des.*, vol. 25, no. 10, pp. 2076–2087, Oct. 2006.

[19] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, "Parallel shortest path algorithms for solving large-scale instances," in *Proc. 9th DIMACS Implementat. Challenge: The Shortest Path Problem*, Nov. 2006, pp. 249–290.

[20] G.-J. Nam, S. Reda, C. J. Alpert, P. G. Villarrubia, and A. B. Kahng, "A fast hierarchical quadratic placement algorithm," *IEEE Trans. Comput.-Aided Des.*, vol. 25, no. 4, pp. 678–691, Apr. 2006.

[21] G.-J. Nam and J. Cong, *Modern Circuit Placement: Best Practices and Results*. New York: Springer, 2007.

[22] M. Pan, N. Viswanathan, and C. Chu, "An efficient and effective detailed placement algorithm," in *Proc. ICCAD*, 2005, pp. 48–55.

[23] S. K. Raman, V. Pentkovski, and J. Keshava, "Implementing streaming SIMD extensions on the Pentium III processor," *IEEE Micro*, vol. 20, no. 4, pp. 47–57, Jul.–Aug. 2000.

[24] J. A. Roy and I. L. Markov, "ECO-system: Embracing the change in placement," *IEEE Trans. Comput.-Aided Des.*, vol. 26, no. 12, pp. 2173–2185, Dec. 2007.

[25] J. A. Roy, D. A. Papa, S. N. Adya, H. H. Chan, A. N. Ng, J. F. Lu, and I. L. Markov, "Capo: Robust and scalable open-source min-cut floorplacer," in *Proc. ISPD*, 2005, pp. 224–226.

[26] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA: SIAM, 2003.

[27] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2: A fast force-directed quadratic placement approach using an accurate net model," *IEEE Trans. Comput.-Aided Des.*, vol. 27, no. 8, pp. 1398–1411, Aug. 2008.

[28] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. Philadelphia, PA: SIAM, 1997, pp. 296–298.

[29] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in *Proc. ASPDAC*, 2007, pp. 135–140.

[30] N. Viswanathan, G.-J. Nam, C. J. Alpert, P. Villarrubia, H. Ren, and C. Chu, "RQL: Global placement via relaxed quadratic spreading and linearization," in *Proc. DAC*, 2007, pp. 453–458.

[31] S. J. Wright, *Primal-Dual Interior-Point Methods*. Philadelphia, PA: SIAM, 1987, pp. 1–309.

Myung-Chul Kim received the B.S. degree in electronic and electrical engineering from the Pohang University of Science and Technology, Pohang, Korea, in 2006, and the M.S. degree in electrical engineering from the University of Michigan, Ann Arbor, in 2009. Currently, he is pursuing the Ph.D. degree from the University of Michigan.

In 2011, he was a Research Intern with IBM Research, Detroit, MI. His current research interests include very large-scale integrated physical design automation with emphasis on placement, routing, and timing analysis.

Mr. Kim is the winner of the ISPD 2010 Clock-Network Synthesis Contest and the recipient of the IEEE/ACM William J. McCalla Best Paper Award at ICCAD 2010.

Dong-Jin Lee was born in Gwangju, Korea. He received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, in 2006, and the M.S. and Ph.D. degrees in electrical engineering from the University of Michigan, Ann Arbor, in 2008 and 2011, respectively.

In 2009 and 2010, he was a Research Intern with Texas Instruments, Dallas. His current research interests include electronic design automation clock network synthesis, and placement and routing algorithms.

Dr. Lee was the winner of the ISPD 2009 and 2010 Clock-Network Synthesis Contests and the recipient of the IEEE/ACM William J. McCalla Best Paper Award at ICCAD 2010.

Igor L. Markov received the Ph.D. degree in computer science from the University of California at Los Angeles, Los Angeles.

He is currently an Associate Professor of electrical engineering and computer science with the University of Michigan, Ann Arbor. He has co-authored three books and more than 180 refereed publications. During the 2011 redesign of the ACM Computing Classification System, he led the effort on the hardware tree. His current research interests include computers that make computers.

Prof. Markov is the recipient of a DAC Fellowship, the ACM SIGDA Outstanding New Faculty Award, the NSF CAREER Award, the IBM Partnership Award, the Microsoft A. Richard Newton Breakthrough Research Award, and the Inaugural IEEE CEDA Early Career Award. He is an Executive Board Member of ACM SIGDA, and Editorial Board Member of the *Communications of ACM* and IEEE Design and Test, as well as several ACM and IEEE transactions. He chaired tracks at DAC, ICCAD, ICCD, DATE, and GLSVLSI. Some of his publications have received the Best Paper Award at the Design Automation and Test in Europe Conference, the International Symposium on Physical Design, and the IEEE Transactions on Computer-Aided Design.