97-48

Quadratic Placement Revisited [1]

C. J. Alpert‡, T. Chan†, D. J.-H. Huang, I. Markov† and K. Yan

UCLA Computer Science Dept., Los Angeles, CA 90095-1596

† UCLA Mathematics Dept., Los Angeles, CA 90095-1555

‡ IBM Austin Research Laboratory, Austin, TX 78758

Abstract

The "quadratic placement" methodology is rooted in [6] [14] [16] and is reputedly used in many commercial and in-house tools for placement of standard-cell and gate-array designs. The methodology iterates between two basic steps: solving sparse systems of linear equations, and repartitioning. This work dissects the implementation and motivations for quadratic placement. We first show that (i) Krylov subspace engines for solving sparse systems of linear equations are more effective than the traditional successive over-relaxation (SOR) engine [15] and (ii) *order convergence* criteria can maintain solution quality while using substantially fewer solver iterations. We then discuss the motivations and relevance of the quadratic placement approach, in the context of past and future algorithmic technology, performance requirements, and design methodology. We provide evidence that the use of numerical linear systems solvers with quadratic wirelength objective may be due to the pre-1990's weakness of min-cut partitioners, i.e., numerical engines were *needed* to provide helpful hints to min-cut partitioners. Finally, we note emerging methodology drivers in deep-submicron design that may require new placement approaches to the placement problem.

1 Introduction

In the physical implementation of deep-submicron ICs, row-based placement, solution quality is a major determinant of whether timing correctness and routing completion will be achieved. In row-based placement, the first-order objective has always been obvious: place connected cells closer together so as to reduce total routing and lower bounds on signal delay. This implies a minimum-wirelength placement objective. Because there are many layout iterations, and because fast (constructive) placement estimation is needed in the floorplanner for design convergence, a placement tool must be extremely fast. As instance sizes grow larger, move-based (e.g., annealing) methods may be too slow except for detailed placement improvement. Due to its speed and "global" perspective, the so-called *quadratic placement* technique has received a great deal of attention throughout its development by such authors as Wipfler et al. [16], Fukunaga et al. [9], Cheng and Kuh [6], Tsay and Kuh [15] and others. Indeed, quadratic placement is reputedly an approach that has been used within commercial tools for placement of standard-cell and gate-array designs.

[1]This work was supported by a grant from Cadence Design Systems Inc.

0-89791-847-9/97/0006/$3.50 DAC 97 - 06/97 Anaheim, CA, USA

This work revisits the quadratic placement methodology and addresses both its mode of implementation and its relevance to future design automatization requirements. The remainder of our paper is as follows. Section 2 develops notation and synthesizes a generic model of quadratic placement. Section 3 briefly summarizes our experiments comparing Krylov-subspace solvers and successive over-relaxation (SOR) solvers. Section 4 proposes a new *order convergence* approach for the quadratic solver. Section 5 analyzes the interaction between the linear system solver and the coupled min-cut partitioning step. Finally, Section 6 discusses the relevance of the quadratic placement approach in the context of past and future algorithmic technology, as well as performance and design methodology requirements.

2 The Quadratic Placement Methodology

2.1 Notation and Definitions

A VLSI circuit is represented for placement by a weighted hypergraph, with n vertices corresponding to modules (vertex weights equal module areas), and hyperedges corresponding to signal nets (hyperedge weights equal criticalities and/or multiplicities). The two-dimensional layout region is represented as an array of legal placement locations. Placement seeks to assign all cells of the design onto legal locations, such that no cells overlap and chip timing and routability are optimized. The placement problem is a form of quadratic assignment; most variants are NP-hard.

Since the numerical techniques used for "quadratic placement" apply only to graphs (hypergraphs with all hyperedge sizes equal to 2), it is necessary to assume some transformation of hypergraphs to graphs via a *net model*. Throughout we use the standard clique model, but we have also obtained similar results for the clique model of [15] as well as for a directed star model.[1]

Definition: The $n \times n$ *Laplacian* $\mathbf{Q} = (q_{ij})$ has entry q_{ij} equal to $-a_{ij}$ for $i \neq j$ and diagonal entry q_{ii} equal to $\sum_{j=1}^{n} a_{ij}$, i.e., the sum of edge weights incident to vertex v_i.

"Pad" constraints fix the locations of certain vertices (typically, due to the pre-placement of I/O pads or other terminals); all other vertices are *movable*. The *one-dimensional placement problem* seeks to place *movable* vertices onto the real line so as to minimize an objective function that depends on the edge weights and the vertex coordinates. The n-dimensional *placement vector* $\mathbf{x} = (x_i)$ gives physical locations of modules $v_1, \ldots, v_n$ on the real line, i.e., x_i is the coordinate of vertex v_i. The corresponding *two-dimensional placement problem* is addressed by the means of independent horizontal and vertical placements.

Squared Wirelength Formulation: Minimize the objective $\Phi(\mathbf{x}) = \sum_{i>j} a_{ij}(x_i - x_j)^2$ subject to constraints $\mathbf{Hx} = \mathbf{b}$. This function can be rewritten as $\Phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{Qx}$.

[1]For a given multipin signal net, the graph edges that represent the net may be constructed in several ways, e.g., a directed star model, an unoriented star model or a clique model (see [2] for a review). The resulting weighted graph representation $G = (V,E)$ of the circuit topology has edge weights a_{ij} derived by "superposing" all derived edges in the obvious manner.

The vast majority of quadratic placers in the literature solve the 2-dimensional placement problem with a top-down approach, i.e., one-dimensional placement in the horizontal direction is used to divide the circuit into left and right halves, after which a placement in the vertical direction is used to subdivide the netlist into quarters, etc.

2.2 Essential Structure of a Quadratic Placer

We now review essential components of the quadratic placement paradigm to establish the historical couplings of numerical optimizations with min-cut optimizations or other means of "spreading out" a continuous force-directed placement. We will illustrate our discussion by referring to the PROUD algorithm of Tsay et al. [15] [14].

Like other works, PROUD considers the squared wirelength objective $\Phi(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T Q\mathbf{x}$. An unconstrained formulation is obtained by considering the objective function $\Phi(\mathbf{x})$ without discrete slot constraints (i.e., a two-dimensional array of allowed locations) for c movable modules, but considering the possibility of f fixed pad constraints. The function to minimize is then written as

$$\begin{aligned}\Phi(\mathbf{x}) &= \frac{1}{2}\begin{bmatrix} \mathbf{x}_c & \mathbf{x}_f \end{bmatrix}\begin{bmatrix} \mathbf{Q}_{cc} & \mathbf{Q}_{cf} \\ \mathbf{Q}_{fc} & \mathbf{Q}_{ff} \end{bmatrix}\begin{bmatrix} \mathbf{x}_c \\ \mathbf{x}_f \end{bmatrix} \\ &= \frac{1}{2}(\mathbf{x}_c^T\mathbf{Q}_{cc}\mathbf{x}_c + \mathbf{x}_c^T\mathbf{Q}_{cf}\mathbf{x}_f + \mathbf{x}_f^T\mathbf{Q}_{fc}\mathbf{x}_c + \mathbf{x}_f^T\mathbf{Q}_{ff}\mathbf{x}_f)\end{aligned}$$

where $\mathbf{x}_f$ denotes the vector of *fixed* module positions and $\mathbf{x}_c$ denotes the vector of *movable* module positions; the Laplacian $\mathbf{Q}$ is partitioned into four corresponding parts $\mathbf{Q}_{cc}$, $\mathbf{Q}_{cf}$, $\mathbf{Q}_{fc}$ and $\mathbf{Q}_{ff}$ with $\mathbf{Q}_{cf}^T = \mathbf{Q}_{fc}$.

Because the optimal positions of all movable modules in quadratic placement are inside the convex hull spanned by the fixed modules [15], we can consider the minimization problem for $\Phi(\mathbf{x})$ over this convex hull. Since $\Phi(\mathbf{x})$ is a strictly convex smooth function over a compact set (in c-dimensional Euclidean space), the unique minimal value is attained at the extremal or a boundary point; the nature of the problem implies that it will be the extremal point. To find the zero of the derivative of the objective function $\Phi(\mathbf{x})$, we solve the $c \times c$ linear system

$$\nabla\Phi(\mathbf{x}) = \mathbf{Q}_{cc}\mathbf{x}_c + \mathbf{Q}_{cf}\mathbf{x}_f = 0$$

which can be rewritten as

$$\mathbf{Q}_{cc}\mathbf{x}_c = -\mathbf{Q}_{cf}\mathbf{x}_f \qquad (1)$$

This development is similar to that of other "force-directed" or "resistive network" analogies (see, e.g., [6] [9] [13] [16]). The essential tradeoff relaxes discrete slot constraints and changes the "true" linear wirelength objective into a squared wirelength objective, in order to obtain a continuous quadratic minimization for which a global optimum can be found. However, the typical resulting "global placement" concentrates modules in the center of the layout region. The key question is how the "global placement" (actually, a "continuous solution obtained using an incorrect objective") should be mapped back to the original discrete problem.

Two approaches have been used to obtain a feasible placement from a "global placement". The first approach is based on *assignment*, either in one step (to the entire 2-dimensional array of slots) or in two steps (to rows, and then to slots within rows) [9]. The second and more widely-used approach is *partitioning*: the global placement result is used to derive a horizontal or vertical cut in the layout, and the continuous squared-wirelength optimization is recursively applied to the resulting subproblems (see [6, 12, 13, 16]). The main difficulty is making partitioning decisions on the extremely overlapped modules in the middle of the layout (see Figure 1). The obvious median-based partitioning (find the median module and use it as a "pivot") is sensitive to numerical convergence criteria. Thus, iterative improvement is commonly used to refine the resulting partitioning (see, e.g., [12]). A typical objective for the iterative improvement is some form of minimum weighted cut. Hence, quadratic placers can be quite similar in structure to top-down min-cut placers, with initial cuts induced from placements under the squared-wirelength objective.

3 Krylov-Subspace Solvers

Recall that quadratic placement solving a series of sparse systems of linear equations. We now review experiments with iterative solvers such as Successive Over-relaxation (SOR/SSOR), BiConjugate Gradient Stabilized (BiCGS) and others (see [4]) showing that the commonly used SOR/SSOR methods (developed in the early 1950's) are not the best available now.

The time complexity of an iterative solver depends on both the cost of a single iteration (which is constant during the solution of a given system) and the number of iterations needed until iterates adequately reflect the true solution. The theory of iterative methods shows that the number of iterations needed to obtain a good approximation in norm depends on the spectrum of the matrix involved [10]. Hence, the idea of a preconditioner – a way to transform the original system to an equivalent one with "improved" spectrum. Because most implementations of preconditioners entail additional per-iteration cost, one must carefully examine the overall efficiency of solver/preconditioner combinations on particular classes of instances: more expensive iterations must be balanced against the number of iterations saved. We have solved a number of test systems with multiple combinations of solvers and preconditioners, and recorded number of iterations as well as CPU usage.[2]

We find that BiConjugate Gradient Stabilized (BiCGS) is among the best solvers; though it does not guarantee convergence, the method is good even for degenerate (not necessarily symmetric) matrices and in our experience provides more robust convergence

[2]See [4] for pseudocodes of solvers and preconditioners that we used and their efficiency comparisons; see [10] for theoretical analyses. A brief review of the relevant numerical methods is as follows. Iterative methods for solving large systems of linear equations can be classified as *stationary* or *non-stationary*. Stationary methods include Jacobi, Gauss-Seidel, Successive Over-relaxation (SOR) and Symmetric Successive Over-relaxation (SSOR). They are older, easier to implement and computationally cheaper per iteration. Non-stationary methods include Conjugate Gradient (CG), Generalized Minimal Residual (GMRes) and numerous variations. These are relatively newer and notably harder to implement and debug, but provide for much faster convergence. Additional computational expense per iteration (sometimes by a factor of 7) is normally justified by much smaller numbers of iterations. Jacobi method solves once for every variable with respect to the other variables. Gauss-Seidel uses updated values as soon as they are computed. SOR is a modification of Gauss-Seidel which depends on the extrapolation parameter ω (finding optimal values of ω is non-trivial, and heuristics are typically applied). SSOR has no computational advantage over SOR as a solver, but is useful as a preconditioner; one iteration is roughly twice as expensive as an SOR iteration.

CG or GMRes will generate a sequence of orthogonal vectors which are residuals of the iterates. CG is very effective when the matrix is symmetric and positive definite, while GMRes is useful for general non-symmetric matrices. Particular modifications of CG include BiConjugate Gradient (BiCG), Conjugate Gradient Squared (CGS), Quasi-Minimal Residual (QMR) and BiConjugate Gradient Stabilized (BiCGS). In BiCG, two sequences are generated which are only mutually orthogonal. The method is useful for non-symmetric nonsingular matrices; convergence is irregular and the method can break down. QMR applies a least-squares update to smoothen the irregularity of BiCG; it is more reliable but rarely faster. Further improvements are given by two variations of Transpose-Free Quasi-Minimal Residual (TFQMR and TCQMR). CGS is a fast method with even more irregular convergence than BiCG, while BiCGS is just another way to smoothen convergence of BiCG. Finally, for positive definite matrices, the Chebyshev iteration computes coefficients of a polynomial minimizing the residual norm. Solvers which provide smooth convergence can be also used as preconditioners. Direct solvers present a different source of preconditioners for iterative methods, with examples being incomplete Cholesky (ICC), LU-factorization and incomplete LU-factorization (ILU).

Solvers		Preconditioners						
		None	SOR/1.0	SSOR/1.0	SOR/1.95	SSOR/1.95	Jacobi	ILU
CG	Iter	245	—	106	—	316	245	97
	CPU	39.69	—	37.24	—	110.50	40.74	34.31
GMRes	Iter	378	180	125	220	380	378	107
	CPU	134.40	79.22	67.43	96.09	206.05	133.12	57.15
BiCGS	Iter	120	74	54	105	128	120	51
	CPU	38.97	36.43	37.33	51.22	88.90	39.30	35.74
CGS	Iter	172	97	50	110	152	172	58
	CPU	55.52	47.57	34.68	53.83	104.91	55.55	40.24
TFQMR	Iter	172	90	48	102	152	172	58
	CPU	59.22	45.75	34.29	52.00	108.64	60.00	41.46
TCQMR	Iter	206	185	94	178	231	206	86
	CPU	115.23	149.47	104.67	143.40	257.37	116.29	93.60

Table 1: Study of solvers and preconditioners for an industry test case with matrix dimension 29308. CPU times are in seconds for a Pentium 150 running Linux 2.0.0; the g++ 2.7.2 compiler was used with -O -m486 options. Dashes (—) indicate that more than 1000 iterations were made, at which point the solver was terminated. The SSOR and SOR preconditioners were run with both $\omega = 1.0$ and $\omega = 1.95$. Identical convergence test and tolerances were used for all runs.

than conjugate gradient (CG). For preconditioners, Incomplete LU-factorization and the Successive Over-relaxation family (including SSOR) are particularly successful. In many tests, the vacuous preconditioner was surprisingly competitive with the best nontrivial preconditioner (iteration count was worse, but iterations were cheaper). At the same time, using the wrong preconditioner could easily lead to a 3-fold loss in CPU time. The relative performance depicted in Table 1 is representative of our results. Table 2 shows the win of using BiCGS over SOR.

Test Case	Size	Nonzeros	BiCGS/SOR, $\omega = 1.0$		SOR, $\omega = 1.95$	
			Iter #	CPU(s)	Iter #	CPU(s)
avq_large	25114	159128	29	10.05	191	29.59
case1	29308	619492	88	76.03	256	98.98
case2	39917	732329	108	115.62	1000+	479.44
golem3	100281	371633	30	54.68	110	95.61

Table 2: Comparable implementations of the SOR and BiCGS (with SOR / $\omega = 1.0$ preconditioner) linear system solvers. tested on a Pentium 150 running Linux 2.0.0. Benchmarks case1 and case2 are matrices supplied from industry. The convergence test and tolerance values are slightly different from those used to generate Table 1 .

4 Order Convergence Criteria

Any solver from the previous section builds a sequence of iterates that converges to the solution **x** of Equation (1). How soon the iteration can be stopped determines performance. Typical convergence tests are based on some norm of the *residual* vector for an iterate[3]; which is taken to represent error with respect to the true solution. In practice, most norms are equivalent; heuristics (check convergence every j iterations, check differences of iterates rather than residual vectors, etc.) can reduce the time spent on convergence tests.

We observe that using a placement solution solely to construct an initial min-cut partitioning solution wastes information: all that is retained are memberships of vertices in "left" and "right" groups. If the final iterate will be sorted and split to induce an initial solution for the min-cut partitioner, then the iteration should terminate when further changes will be inessential to the partitioner. This depends on the strength and stability of the partitioner, but the iteration should at least stop when the left and right groups stabilize. We now define several heuristic alternatives for what we call *order convergence* criteria. Consider an iterate $\mathbf{x_k}$ from some linear system solver, whose i-th coordinate $x_k(i)$ gives the location of module v_i at this iteration. The placer relies on the relative sorted order of the coordinates, rather than their absolute values, to assign modules to sub-blocks. We use a *direct permutation* π_k^+ to represent the ordering induced by the k-th iterate. If v_i is the j-th module in the ordering induced by $\mathbf{x_k}$, then $\pi_k^+(j) = i$; $\pi_k^-(i) = j$ defines the the *inverse permutation* π_k^- (see Table 3).

[3]When solving the system $\mathbf{Ax} = \mathbf{b}$, the residual vector for a given iterate $\mathbf{x_k}$ is $\mathbf{b} - \mathbf{Ax_k}$.

Index	0	1	2	3	4	5	6	7	8	9	10
Iter $\mathbf{x_k}$	20	16	12	1	-99	8	23	99	1	11	-3
Iter $\mathbf{x_{k+1}}$	20	16	12	1	99	8	23	-99	1	11	-3
Direct permutations											
π_k^+	8	7	6	2	0	4	9	10	3	5	1
π_{k+1}^+	8	7	6	2	10	4	9	0	3	5	1
Inverse permutations											
π_k^-	4	10	3	8	5	9	2	1	0	6	7
π_{k+1}^-	7	10	3	8	5	9	2	1	0	6	4

Table 3: Direct (π_k^+) and inverse (π_k^-) permutations for two 11-dimensional iterates $\mathbf{x_k}$ and $\mathbf{x_{k+1}}$

In computation of π_k^+ and π_k^-, real values are considered equal if they are ε-close (for a predefined $\varepsilon \geq 0$). With this addition, π_k^+ and π_k^- are not well-defined unless the sorting algorithm is stable. A convergence measure based on permutations is convenient for theoretical analysis, but we also seek linear-time implementations (e.g. with k^{th}-largest element finding). Some intuition is given by:

Order Convergence Theorem Given $\mathbf{x_k} \to \mathbf{x}$, π_k^+ and π_k^- converge if computed with sufficiently small $\varepsilon > 0$.

Proof Without loss of generality assume convergence in the L^∞-norm. Choose ε smaller than $\frac{1}{2}$ of the smallest distance between two distinct limit values for coordinate slots. Then, starting with some N-th iteration, each coordinate should be $\frac{\varepsilon}{2}$-close to its limit value. Consequently, the values in two coordinate slots will either differ by more than ε or will be ε-close (if the limit values are equal). $\square$

Note that if the distance between two limit coordinate values is precisely ε, π_k^+ and π_k^- may never become constant. This shows that it may be computationally difficult to reach complete order stabilization if limit values of many coordinates are close (which is the case with quadratic placement; see Figure 1). Rather than try to circumvent this phenomena, we intend to use it (see below).

We have worked with several order convergence measures, which include the following (due to space limitations, we do not give motivations, but only constructions of certain measures). An important building block in our construction is the maximum change in sorted index that any module experiences between iterates $\mathbf{x_k}$ and $\mathbf{x_{k+1}}$:

- **max**D^{++} $= max_{0 \leq i \leq N}\{|\pi_k^+(i) - \pi_{k+1}^+(i)|\}$

  This is the maximal placewise difference between two direct permutations, i.e., for each coordinate slot we compute the two indices to which the values sort in consecutive iterates, then take the maximum difference of these two indices over all slots. In Table 3, max$D^{++} = 10$ since coordinate slot 4 has value 99 (index 10) in one iterate and value -99 (index 0) in the other iterate.

In practice, a min-cut partitioner may be biased by locking modules that are at extreme indices in the sorted ordering. This decreases the size of the solution space and reduces runtime. Alternatively, modules might be left unlocked, but the starting solution constructed to capture extreme module coordinates (see the GORDIAN methodology, for example [12]). Hence,

- $\mathbf{maxD^{++}_{10\%}} = max_{0 \leq i \leq \frac{N}{10};\ \frac{9N}{10} \leq i \leq N}\{|\pi_k^*(\pi_k^-(i)) - \pi_{k+1}^*(\pi_k^-(i))|\}$

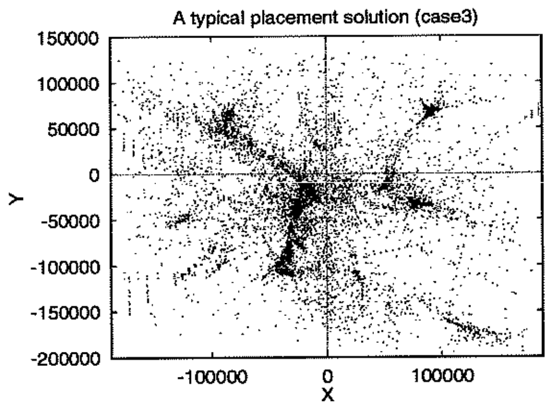

Figure 1: Two-dimensional global placement obtained by minimizing squared wirelength separately in the x and y directions. This example (case3) has 12146 cells and 711 I/O pads fixed on four sides of the layout. Modules are concentrated in small regions, and we use this feature to formulate *order convergence* tests.

One may compute maximal differences over only some percentage (in these examples, 20%) of the indices. In this example, we identify places with 10% smallest and 10% largest values and check how far these values "travel" (in terms of ordering index).

Finally, we have studied the measures

- $\mathbf{Flow}^{\%\%}_{\mathbf{10\%}} =$
  $\#\{i | (0 \le i \le \frac{N}{10} \textbf{ or } \frac{9N}{10} \le i \le N) \textbf{ and } \frac{N}{10} \le \pi^{+}_{k+1}(\pi^{-}_{k}(i)) \le \frac{9N}{10}\}$
- $\mathbf{Flow}^{\mathbf{Num}}_{\mathbf{10\%}} = \#\{i | (0 \le i \le \frac{N}{10} \textbf{ or } \frac{9N}{10} \le i \le N)$
  $\textbf{and } x_k(\pi^{-}_{k}(\frac{N}{10})) + \varepsilon \le x_{k+1}(\pi^{-}_{k}(i)) \le x_k(\pi^{-}_{k}(\frac{9N}{10})) - \varepsilon\}$
  Here, we use %% to denote "percent", and **Num** to denote "number of". The basic idea is to compute the "flow" of coordinates through some "percentage barrier". For example, $\mathbf{Flow}^{\%\%}_{\mathbf{10\%}}$ computes how many of the lowest 10% and highest 10% of coordinates leave this range in the next iteration. $\mathbf{Flow}^{\mathbf{Num}}_{\mathbf{10\%}}$ detects the *numerical* range of the lowest 10% and highest 10% in the iterate $\mathbf{x_k}$ (i.e., two intervals $[min, a], [b, max]$) and computes how many coordinates leave this numerical range in the next iteration. In this case, we allow for absolute error ε.

Our intuition behind the use of such order convergence measures is as follows. During early iterations the order of coordinates is expected to change because the initial guess has very little in common with the true solution. These order changes will decrease as the iterates approach the solution. However, when the iterates are *very close* to the true solution, modules will become concentrated in small areas, and order changes will *increase* due to small displacements of many closely located vertices, which should be immaterial to a strong partitioner. Therefore, one should stop iterations when order convergence measures start increasing, or earlier if they become close enough to zero.

5 Experimental Results

We now present experimental data which yields insights regarding the proper "solver-partitioner interface" (i.e., the point at which solver iterations should be stopped, and an instance fed to the min-cut partitioner). Our examples are derived from the benchmark netlists avq_large (25114 cells, 25144 nets and 87017 pins) and from an industry-supplied test case, Case3 (12857 cells, 10880 nets and 58874 pins).[4] Matrices were obtained as described in Section 2.2 for the x-direction ordering problem with standard clique model and thresholding of nets with > 99 pins. All iterates were saved from both SOR and BiCGS runs with convergence criterion being residual norm $< 10^{-4}$.

5.1 Performance Gains

Figure 2 presents order convergence measures between consecutive solver iterates for examples case3 and avq_large with respect to a flow-based measure. One sees three[5] distinct periods in the convergence history, which we interpret as follows:

- **Rapid Order Convergence** The order of coordinates (equal to each other in the initial guess) changes rapidly, approaching that of the true solution.
- **Coordinate Adjustment** The order stabilizes while coordinate values still change to approach those of the true solution
- **Coordinate Refinement** Vertices become clustered in small regions, and small changes in their coordinates (immaterial to partitioner) produce peaks of order convergence scores.

As seen in Figure 2, most of the gain from solving a linear system is achieved during first 10-15 iterations. The remaining CPU time (until convergence is signaled by traditional critera, based on some *residual norm*) can and should be used elsewhere.

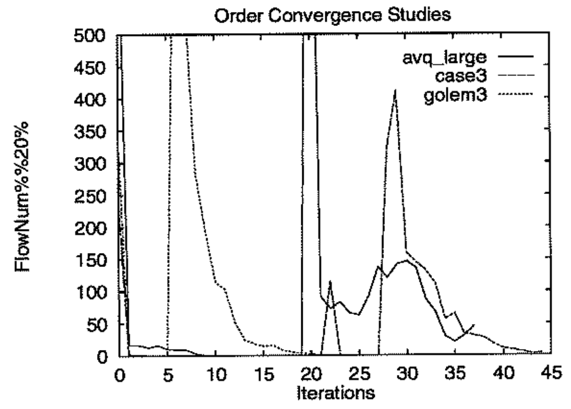

Figure 2: Order convergence studies for examples case3, avq_small and golem3(scaled by 0.1) with SOR iterates. From each 5 consecutive order convergence scores computed with $\mathtt{Flow}^{\mathrm{Num}}_{\%\%}20\%$, the best 3 have been averaged. $\mathtt{Flow}^{\mathrm{Num}}_{\%\%}20\%$ used error tolerance 10^{-4} times the size of placement interval. The plot shows that the order of coordinates stabilizes quickly.

[4]We also performed experiments with golem3 (103048 cells, 144949 nets and 339149 pins), but we can not include results due to space considerations. The behavior observed with golem3 is similar to that observed with case3 and avq_large (see [3] for the complete set of experiments). Both avq_large and golem3 are available from http://www.cbl.ncsu.edu (the CAD Benchmarking Laboratory). See also our website, http://vlsicad.cs.ucla.edu/ .

[5]The *Order Convergence Theorem* above suggests that for small enough $\varepsilon > 0$, there will be a fourth period of **Final Convergence** in which the order becomes constant.

5.2 Interaction with the Partitioner

We analyzed SOR iterates for each of our test cases by sorting the coordinates and then inducing initial solutions for a Fiduccia-Mattheyses (FM) [8] min-cut partitioner. Our experimental procedure was as follows. (1) Initial solutions were induced by pre-seeding some percentage of leftmost and rightmost vertices in the ordering into the initial left and right partitions, respectively. Remaining vertices were assigned randomly into the initial left and right partitions. Three pre-seeding percentages of 0%, 20% and 50% were used. A pre-seeding of 0% corresponds to an initially random solution, and 50% corresponds to the solution obtained by splitting the iterate. (2) All vertices were free to move except for fixed pads which were locked according to whether they were to the left or right of the median coordinate. (3) For each iterate, we generated multiple pre-seeded initial solutions and ran FM from each, using unit module areas and an exact bisection requirement.

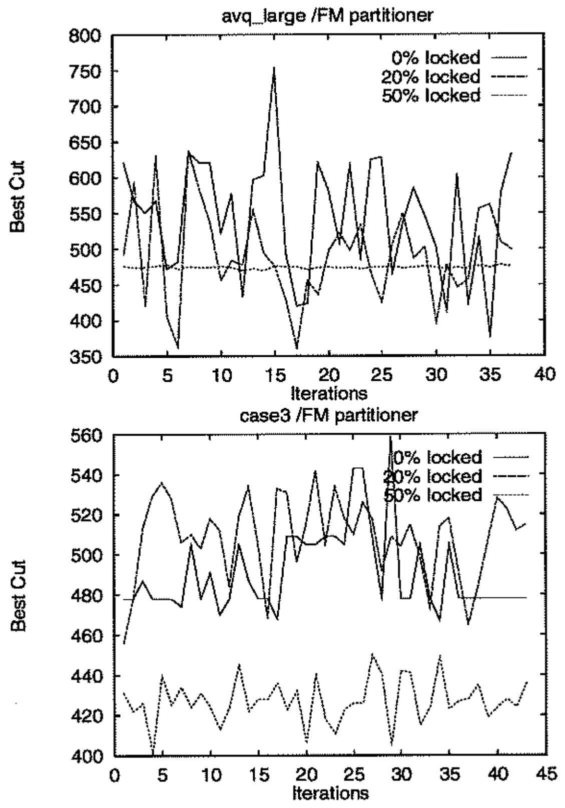

Figure 3: Iteration number versus minimum cut obtained over 30 runs of FM using the iterate as a pre-seed. Using 50% pre-seeded modules yields large improvements.

Figure 3 shows the minimum cut obtained from 30 pre-seeded solution as a function of the SOR iterate. It is clear that strong (50%) pre-seeding enables FM to return better solutions than using initially random solutions (0%) or somewhat locked solutions (20%). Note that the benefit of using a later iterate as opposed to a relatively early iterate is marginal; there is indeed a potential gain if we can apply order convergence criteria. We have also observed that not only do the cutsizes remain similar from iterate to iterate, but the solutions themselves do not vary significantly (see [3] for more details, including data showing Hamming distances for various pairs of partitioning solutions).

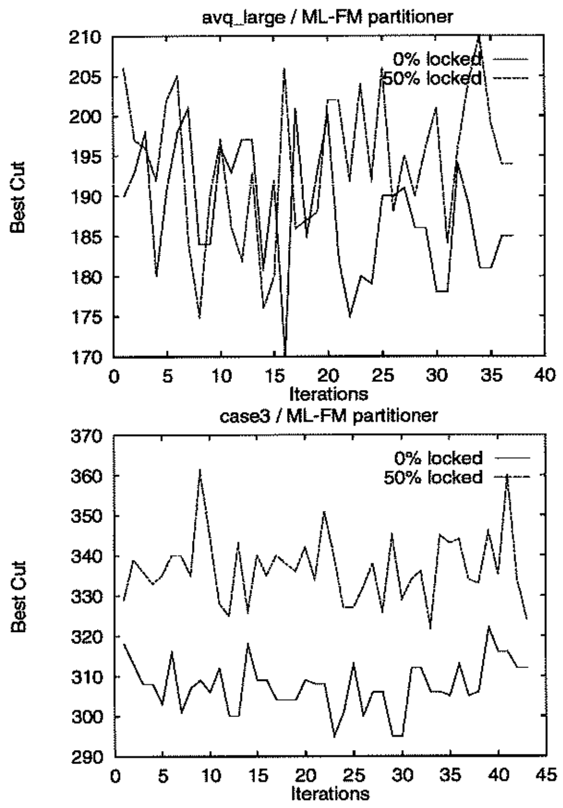

Figure 4: Iteration number versus minimum cut obtained for 5 runs of ML-FM using the iterate as a pre-seed. The 0% locked solutions clearly outperform the 50% solutions. 20% solutions are not shown since they are significantly worse than both 0% and 50% solutions.

5.3 Modern Partitioners Do Not Need Hints

The previous experiments confirm that a traditional FM min-cut partitioner can certainly benefit from the hints contained in solver iterates. We now propose an interesting notion; that the use of numerical linear systems solvers with quadratic wirelength objective is historically due to the pre-1990's weakness of min-cut partitioners. In the past few years, significant improvements to FM have been made, primarily in the areas of tie-breaking and multilevel schemes (see [2] for a survey). The multilevel approach integrates hierarchical clustering into FM, generalizing the early "two-phase" approach (e.g., [5]). The recent work of [1] developed a multilevel partitioner that reports outstanding solution quality with respect to many other methods in the literature. We have obtained this partitioning code and integrated it into our testbed.

We repeated the previous set of experiments using 5 runs of multilevel (ML-FM) [1] instead of FM partitioning. Figure 4 shows how ML-FM solution quality varies with convergence of SOR, and with the amount of information retained from the iterate.[6] The 20% data was omitted for these plots since the cuts were much worse than the 0% or 50% data (likely due to ML-FM's inability to handle pre-seeding in a natural way). The conclusions are clear: ML-FM dramatically outperforms FM, and furthermore draws no benefit

[6]Given an initial bipartitioning solution and an 0.5 level of pre-assignment, ML-FM determines a bottom-up clustering that is compatible with this solution throughout the hierarchy. With an 0.0 level of pre-assignment, ML-FM has no constraints on its bottom-up clustering. From the results in Figure 4, these constraints actually hurt solution quality; other pre-seeding techniques for guiding ML-FM should be explored.

from using solver iterates. In fact, ML-FM solutions are generally worse when constrained to follow the structure of an iterate in its initial solution. We have also observed that the ML-FM solutions for different iterates are extremely similar in terms of both structure and cut cost, no matter what initial solution is chosen. Thus, that if a minimum-cut solution is desired, a viable approach may be to use ML-FM on a random initial solution, and not use any quadratic placement techniques.

Our experimental data leads to the surprising hypothesis that a linear system solver may be completely avoided in the quadratic placement approach with no loss of placement quality. While strong partitioners "ignore" hints from the linear system solver, it is certain that a partitioner is needed to counteract the effects of clumping and squared wirelength objective. We by no means suggest that placement reduces to partitioning *on one level*, but rather that such is the case in the *top-down* context, where rich geometric information is implicit in the partitioning instance.

6 Discussion and Futures

We have synthesized the motivations and structure for a generic "quadratic placement" methodology, and given both performance improvements and a historical context for the approach. We have also shown that – possibly – an implementation of the approach no longer requires an embedded linear systems solver, and indeed can revert back to "min-cut placement" when armed with the latest partitioning technology. We by no means claim that every existing quadratic placer should discard its numerical engine, but – all else being equal – we suspect that if a minimum weighted cut is the true objective, ML-FM or other recent partitioners might be invoked with no loss of solution quality.[7]

More generally, we believe that there are basic drivers that suggest looking beyond "quadratic placement" technology. First, there are well-known limitations to modeling capability within a quadratic placer, e.g., path timing constraints, invariance of orderings to unequal horizontal and vertical routing, the requirement of pre-placed pads to "anchor" the placement, etc. Second, a top-down, performance- and HDL-driven design methodology will have relatively smaller technology-mapped blocks in order to gain predictability; these will not be large enough for a quadratic placer to show its "global awareness" and runtime advantages. Third, the advent of block-based design may mean fewer large, flat problem instances. Synthesized glue logic will be spread out over disconnected, heterogeneous regions; the resulting chip planning - block building - assembly flow harkens back to classic block packing and route planning issues, and does not play to the strengths of quadratic placement. Thus, while quadratic placers are the state of the art today, it remains to be seen whether other approaches will more effectively address future placement requirements.

REFERENCES

[1] C. J. Alpert, D. J.-H. Huang and A. B. Kahng, "Multilevel Circuit Partitioning", to appear in *DAC*, 1997.

[2] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration, the VLSI Journal*, 19(1-2) (1995), pp. 1-81.

[3] C. J. Alpert, T. Chan, D. J.-H. Huang, I. Markov and K. Yan, "Quadratic Placement Revisited", *Technical Report #970012*, UCLA Computer Science Dept., March 1997.

[4] R. Barret, M. Berry, T. Chan et al "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", *SIAM* 1995, `http://netlib2.cs.utk.edu/linalg/html_templates/Templates.html`

[5] T. Bui, C. Heigham, C. Jones and T. Leighton, "Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms", *DAC* (1989) pp. 775-778.

[6] C. K. Cheng and E. S. Kuh. "Module Placement Based on Resistive Network Optimization", *IEEE Trans. on CAD* 3 (1984), pp. 218–225.

[7] H. C. Elman, M. P. Chernesky, "Ordering Effects on Relaxation Methods Applied to The Discrete One-Dimensional Convection-Diffusion Equation", *SIAM J. Numer. Anal.*, 30(5) (1993), pp.1268-1290.

[8] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions", *DAC* (1982), pp. 175-181.

[9] K. Fukunaga, S. Yamada, H. S. Stone, and T. Kasai, "Placement of Circuit Modules Using a Graph Space Approach", *Proc. 20th ACM/IEEE Design Automation Conference*, (1983), pp. 465–471.

[10] W. Hackbush, "Iterative Solution of Large Sparse Systems" Springer Verlag, 1994.

[11] L. W. Hagen, D. J.-H. Huang, and A. B. Kahng, "Quantified Suboptimality of VLSI Layout Heuristics", *DAC* (1995), pp. 216-221.

[12] J. Kleinhans, G. Sigl, F. Johannes and K. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization", *IEEE Trans. on CAD.* 10(3) (1991) pp. 356-365.

[13] R. H. J. M. Otten, "Automatic Floorplan Design", *DAC* (1982), pp. 261-267.

[14] R. S. Tsay, E. Kuh and C. P. Hsu, "Proud: A Sea-Of-Gate Placement Algorithm", *IEEE Design & Test of Computers* (1988), pp. 44-56.

[15] R. S. Tsay and E. Kuh, "A Unified Approach to Partitioning and Placement", *IEEE Trans. on Circuits and Systems*, 38(5) (1991), pp. 521-633.

[16] G. J. Wipfler, M. Wiesel and D. A. Mlynski, "A Combined Force and Cut Algorithm for Hierarchical VLSI Layout, *DAC* (1983), pp. 124-125.

[7]The reader may observe that we have used only "top-level" instances to test our hypotheses. We chose to use these instances because they have the fewest "pad" constraints and thus the greatest leeway within which a partitioner might benefit from solver hints. (One might argue that when there are relatively many "pad" constraints due to terminal propagation and Rent's rule, the benefits from a strong partitioner will decrease. This is true, but the benefits from solver hints will also decrease correspondingly in such a situation.) The reader may also observe that extremely good min-cuts can under-utilize routing resources across the cut line. The fact remains that quadratic placement approaches in the literature do attempt to minimize cut size when defining hierarchical subproblems. If a quadratic placer were to *benefit* from having a bad partitioner, this would of course bring into question both the objective and the role of partitioning in quadratic placement.