Seeing the Forest and the Trees: Steiner Wirelength Optimization in Placement

Jarrod A. Roy, James F. Lu and Igor L. Markov
The University of Michigan, Department of EECS
1301 Beal Ave., Ann Arbor, MI 48109-2122
{royj, jflu, imarkov}@eecs.umich.edu

ABSTRACT

We show how to optimize Steiner-tree Wirelength (StWL) in global and detail placement without a significant runtime penalty, making the use of Half-Perimeter Wirelength unnecessary. Given that StWL correlates with Routed Wirelength (rWL) much better than HPWL, our new optimization improves the overall Place-and-Route results. We also develop congestion-driven whitespace distribution during global placement. Comparing to the state of the art where whitespace is redistributed after global placement, we demonstrate that cell-shifting techniques often increase via counts.

Our placer ROOSTER outperforms best published results for Dragon, Capo, FengShui, mPL-R/WSA and APlace in terms of routed wirelength by 10.7%, 5.6%, 9.3%, 5.5% and 4.2% respectively. Via counts, especially important at 90nm and below, are improved by 15.6% over mPL-R/WSA and 11.9% over APlace.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-Aided Design

General Terms
Algorithms, Design

Keywords
Physical Design, Placement, Routing, Steiner Tree

	Objectives/constraints in Place-and-Route	Use in placement: Pertinent	Popular	Ours	Our empirical improvements
Relative	Routability	*			+
	Routed WL	*			+
	Via count	*		limit	+
	Timing	*	~	potential	
	Dynamic power	*		potential	
	Router runtime	*			+
Absolute	Congest estimates	?	*	*	+
	Placer runtime	*	*	limit	-
	Steiner-tree WL			*	+
	HPWL		*		-

Table 1: Traditional work on placement does not optimize or even report the objectives most pertinent for Place-and-Route. It is particularly difficult to optimize objectives that are measured *relative* to a given industrial router. We improve key objectives by departing from the traditional HPWL optimization. (?) Optimizing congestion *per se* appears of limited use.

1. INTRODUCTION

Recently there has been much interest in estimating the amount of improvement that is left in placement optimization [6]. The gap between *optimal* and *practically achievable* solutions is usually explained by the difficulty of optimization and shortcomings of individual algorithms. In this work we point out another major source of sub-optimality in Physical Design — *solving wrong optimization problems*, whether optimally or not. In the short term, this source of sub-optimality seems fairly easy to address, as confirmed by our empirical results. We improve placement algorithms by leveraging existing research on minimal Steiner trees.

Our main contribution is optimizing Steiner-tree Wirelength (StWL) in global and detail placement without a significant runtime penalty, making the use of Half-Perimeter Wirelength unnecessary. We draw on recent works in min-cut placement, and particularly the terminal propagation technique from [21], improved in [7], which better correlates small net-cut with small HPWL. We generalize this technique and show that with adequate data structures it reduces StWL in global placement efficiently. To our knowledge, minimization of StWL in min-cut bisection has not been attempted before, particularly the net-vector technique [12] cannot capture Steiner-tree lengths in bisection or quadrisection (for more details see Section 2.1). Our Steiner-tree driven detail placer leverages the speed of the recent FLUTE package [8]. The closest work in detail placement [13] models single-trunk Steiner trees to reduce congestion in FPGAs. While effective, this technique requires exorbitant amounts of runtime. Instead, our detail placer is quite fast, considers optimal Steiner trees, and optimizes a different objective.

We also build up on recent work in congestion-driven placement that uses congestion maps. In [25] congestion maps are built after global placement, and annealing moves are applied to minimize a congestion metric. Another technique, proposed in [18] and known as WSA, is applied after detail placement. It identifies areas with high congestion and injects whitespace into these areas in a top-down fashion. Our work uses congestion maps from [24] to allocate whitespace in a manner similar to WSA but *proactively, during global placement*. As a result, our placer ROOSTER (Rigorous Optimization Of Steiner-Trees Eases Routing) produces the best known routed wirelengths on the IBMv2 benchmarks [25].

At the 90nm technology node and below, increased via resistance, manufacturing variability and manufacturing defects require unprecedented attention to vias. Most semiconductor manufacturers require vias to be doubled for reliability. To this end, we point out that a range of easy-to-implement detail placement algorithms (those of the cell-shifting variety) tend to increase via counts, even when they improve routability. ROOSTER avoids them and exhibits the smallest via counts on standard benchmarks among all published results and our runs of recent placement tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISPD'06, April 9–12, 2006, San Jose, California, USA.
Copyright 2006 ACM 1-59593-299-2/06/0004 ...$5.00.

In the remainder of this paper, Section 2 describes previous work on VLSI placement. Section 3 discusses the realization of Steiner-tree modeling in min-cut placers and Section 4 describes Steiner-driven detail placement. Section 5 outlines whitespace allocation to improve routability. Experimental results are given in Section 6, and Section 7 outlines further applications of our techniques.

2. BACKGROUND AND PREVIOUS WORK

Traditionally, placement and routing are treated as two separate and independent optimization problems. Standard-cell placement is generally seen as the problem of finding non-overlapping row and site-aligned positions for cells while minimizing the wirelength of the design. Currently, HPWL is the estimate of choice for wirelength minimization in placement because it is computationally easy and exactly estimates Rectilinear Steiner Minimal Tree (RSMT) length for 2- and 3-pin nets. Unfortunately, routers construct routed wires using Steiner trees whose length is under-approximated by HPWL. Since RSMT construction is an NP-complete problem [11], it has been generally regarded as too computationally demanding for use in placement [12]. To illustrate how a placer optimizes its chosen objective, we describe a specific placement technique – top-down placement.

2.1 Top-down placement

Top-down placement algorithms seek to decompose a given placement instance into smaller instances by subdividing the placement region, assigning modules to subregions and cutting the netlist hypergraph [3]. Min-cut placers generally use either bisection or quadrisection to divide the placement area and netlist. The netlist division step is commonly implemented with the Fiduccia-Mattheyses heuristic and derivatives [10, 4], or alternatively with quadratic placement and geometric partitioning [2].

Placement bins. Each hypergraph partitioning instance is induced from a rectangular region, or bin, in the layout. In this context a *placement bin* represents (i) a placement region with allowed module locations (*sites*), (ii) a collection of circuit modules to be placed in this region, (iii) all signal nets incident to the modules in the region, and (iv) fixed cells and pins outside the region that are adjacent to modules in the region (*terminals*). Top-down placement can be viewed as a sequence of passes where each pass examines all bins and divides some of them into smaller bins. These smaller bins collectively contain the entire layout area and cells of the original instance. When placement bins are divided, careful choice of vertical or horizontal cut direction influences wirelength and routing congestion in resulting placement solutions [23].

Terminal propagation. Proper handling of terminals is essential to the success of top-down placement approaches. When a particular placement bin is split into multiple subregions, some of the cells inside may be tightly connected to cells outside of the bin. Ignoring such connections can adversely affect the quality of a placement since these connections can account for significant amounts of wirelength. On the other hand, these terminals are irrelevant to the classic partitioning formulation as they cannot be freely assigned to partitions. A compromise is possible by using an extended formulation of "partitioning with fixed terminals", where the terminals are considered to be fixed in ("propagated to") one or more partitions, and assigned zero areas (original areas are ignored). Nets which are propagated to both partitions in bi-partitioning are considered "inessential" since they will always be cut and can be safely removed from the partitioning instance to improve runtime [5]. Terminal propagation [9, 22, 12, 5] is typically driven by geometric proximity of terminals to subregions/partitions.

Minimizing HPWL through weighted net-cut. The authors of [21] also note the inaccuracy of representing the wirelength objective of placement by the min-cut objective in partitioning. Optimizing HPWL directly through partitioning can provide improvements over the simple min-cut objective. The authors introduce a new terminal propagation technique in their placer THETO that allows the partitioner to better map net-cut to HPWL. The terminal propagation in THETO differs from traditional terminal propagation in that each original net may be represented by one or two nets in the partitioned netlist, depending on the configuration of the net's terminals. Two special cases — nets with no terminals and inessential nets — are treated the same as in traditional terminal propagation. Five other cases are analyzed in [21], based on the configuration of terminals relative to the centers of the child bins, and proper weight computation is described (one case requires two nets). This way weighted net-cut better represents the "HPWL degradation" seen after partitioning. Empirically, this terminal propagation and net weighting are shown to reduce HPWL in min-cut placement.

This technique is simplified in [7] and reduced to the calculation of three wirelengths per net per partitioning instance (see more details in Section 3). Our key observation is that this calculation is sufficiently general to facilitate the minimization of wirelength estimates other than HPWL.

Using multi-way partitioning. In an attempt to improve basic recursive bisection, many researchers have noted that it eventually produces multi-way partitions which could be alternatively achieved by direct methods using wirelength-like multi-way objectives. In [12] the authors make use of quadrisection and show how several different cost functions other than cut can be optimized efficiently, although with overhead greater than that of bisection. One such cost function is the Minimum Spanning Tree (MST) length which they note is a far more accurate predictor of routed wirelength than net-cut. The authors note that for a wirelength evaluator to be feasible for placement optimization, it must have evaluation complexity similar to or smaller than MST. On the other hand, the authors claim that their techniques can apply to "arbitrarily complicated per-net placement objectives" [12].

The net-vector technique includes the computation of 2^p integer costs per optimization objective defined for p partitions ($p = 4$ in [12] because quadrisection is used). It then looks up these costs during partitioning. Unfortunately, such look-ups require the discretization of pin locations and cannot account for the location of fixed terminals with as much precision as our work. Furthermore, the Steiner-tree objective on a discretized 2x2-grid does not differ from the discretized MST objective, hence it appears that optimizing StWL would require at least 16-way partitioning, with fairly large net-vector tables. However, no 16-way *geometric* partitioners can be found in the literature that are competitive to recursive bisection. In our work, Steiner trees are built on the fly for each configuration, but the overall runtime remains reasonable.

2.2 Estimating congestion and routed wirelength

Congestion Maps. There have been many recent advances in estimating routing congestion. Most have come in the form of more accurate and faster congestion maps [15, 24]. In this work, we make use of the congestion mapping techniques presented in [24] which assumes that routers attempt to route nets with the fewest number of bends possible. Thus it models two-pin nets in only L and Z shapes, unlike other methods that consider all possible shortest paths between two pins equally. Empirically the authors of [24] have found that some routers are able to find routes with one bend 60% of the time and two bend routes for the majority of other

nets. Thus, one-bend and two-bend routes are weighted this way in their maps. Empirical results show that such estimates correlate well with actual routing usage in the Magma Place-and-Route flow.

Rectilinear minimal Steiner-tree evaluators. The problem of constructing Rectilinear Steiner Minimal trees is known to be NP-hard [11]. Specifically, it is the problem of connecting a given set of points in the Manhattan plane by a minimum-length tree, which can use additional branching (Steiner) points. This problem admits polynomial-time approximations and practical heuristics. Three such algorithms with available source code are Batched Iterated 1-Steiner (BI1ST) [14], FastSteiner [16] and FLUTE [8]. BI1ST is the oldest and slowest of these algorithms but generally produces the best solutions overall. FLUTE is the most recent, the fastest, and provably optimal for instances with nine points or fewer. FastSteiner is in the middle in terms of both speed and solution quality.

2.3 Achieving Routable Placements

It is well-known that a placement with small HPWL may be unroutable due to uneven routing demand and ensuing wiring congestion. For this reason, modern placers must explicitly account for routing congestion in order to produce routable placements. In [25], congestion maps are built after global placement, and annealing moves are applied to minimize a congestion metric. Another technique known as WSA [18] is applied after detail placement. WSA uses congestion maps to identify areas with high congestion and injects whitespace into these areas in a top-down fashion. After all the whitespace allocation is done, cells generally overlap each other and legalization is required. After legalization, window based detail placement techniques are applied to reduce wirelength that was increased during whitespace allocation and legalization. Cell bloating [20] and cell spreading [18] are used to tie whitespace to specific cells, rather than to fixed regions as in techniques based on congestion maps.[1]

3. MINIMIZING TOTAL STEINER-TREE LENGTH IN GLOBAL PLACEMENT

In this section, we describe new techniques to minimize Steiner wirelength in min-cut placement. In addition to the overall methods that make minimizing Steiner wirelength possible, we present data structures new to min-cut placement that keep runtimes practical. These global placement techniques alone can reduce routed wirelength by up to 7%, as demonstrated in Figure 2.

A framework for minimizing StWL. To minimize total StWL during min-cut placement, we capture it using the weighted net-cut objective used in partitioning. In the case of HPWL minimization, this has been accomplished in [21] with a 7-case analysis. A different group reduced this technique to the calculation of three wirelengths per net per partitioning instance [7] and verified resulting empirical improvements.

While the formulation from [7] is more compact than the one from [21], we also note that it is far more general. For each net in each partitioning instance, one must calculate the cost of all nodes on the net being placed in partition 1 (w_1), the cost of all nodes on the net being placed in partition 2 (w_2) and the cost of all nodes on the net being split between partitions 1 and 2 (w_{12}). Up to two nets can be created in the partitioning instance, one with weight $|w_1 - w_2|$ and the other with weight $w_{12} - max(w_1, w_2)$. The only assumption made in [7] is that $w_{12} \geq max(w_1, w_2)$.

[1] Cell bloating artificially increases the *width* of cells because their heights are determined by rows. However, the peak demand for horizontal tracks does not decrease because cells are not spread vertically. To the contrary, by spreading cells horizontally cell bloating increases the overall demand for horizontal tracks.

The points required to calculate w_1 for a given net are the terminals on the net plus the center of partition 1. Similarly, the points required to calculate w_2 are the terminals plus the center of partition 2. Lastly, the points to calculate w_{12} are the terminals on the net plus the centers of both partitions. Clearly the HPWL of the set of points necessary to calculate w_{12} is at least as large as that of w_1 and w_2 since it contains an additional point. By the same logic, StWL also satisfies this relationship since RSMT length can only increase with additional points. Since StWL is a valid cost function for these weighted partitioning problems, we have a framework whereby it can be minimized. To our knowledge, such a framework has not been known in min-cut placement until now.

The simplicity of this framework for minimizing StWL is deceiving. In particular, the propagation of terminal locations to the current placement bin and the removal of inessential nets [5] — standard techniques for HPWL minimization — cannot be used when minimizing StWL. Moving terminal locations drastically changes Steiner-tree construction and can make StWL estimates extremely inaccurate. Nets that are considered inessential in HPWL minimization (where the bounding box of terminals spans the centers of child bins) are not necessarily inessential when considering StWL because there are many Steiner trees of different lengths that have the same bounding box. Not only computing Steiner trees, but even traversing all relevant nets to collect all relevant point locations can be very time-consuming. Therefore, the main challenge in supporting StWL minimization is to develop efficient data structures and limit additional runtime during placement.

Pointsets with multiplicities. Building Steiner trees for each net during partitioning is a computationally expensive task. Similarly, Table 8 in Appendix A shows how expensive a naive replacement of HPWL with Steiner-tree evaluation can be in floorplanning. To keep the runtime reasonable when building Steiner trees for partitioning, we propose a simple yet highly effective data structure — *pointsets with multiplicities*. For each net in the hypergraph, we maintain two lists. The first list contains all the unique pin locations on the net that are fixed. A fixed pin can come from sources such as terminals or fixed objects in the core area. The second list contains all the unique pin locations on the net that are movable, i.e., all other pins that are not on the fixed list. We maintain a unique list of points so that we don't pass any redundant points to Steiner evaluators which may increase their runtime. To do so efficiently, we keep the lists sorted. For both lists, in addition to the location of the pin, we keep the number of pins that correspond to a given point. Before legalization in detail placement, there may be significant cell overlap which can cause pins to have the same location. Maintaining the number of real pins that correspond to a point in a pointset (i.e., the multiplicity of that point) is necessary for efficient update of pin locations during placement.

Steiner weighted min-cut step by step. At the beginning of min-cut placement, all the movable cells are placed at the center of the first placement bin which encompasses the core area. Next all the fixed and movable pointsets are initialized. The runtime required to initialize each pointset is asymptotically the same as sorting the pointset. The multiplicities associated with each point are updated in constant time when duplicates are removed.

When a partitioning instance is to be built for a bin, all nets that are incident to the bin must be examined in any min-cut placer. Usually any cell that is outside of the bin would be propagated to the border of the bin. We skip this step as this reduces the accuracy of the Steiner measurements. Instead we collect all the locations of terminals on this net. This includes all the fixed pins in addition to any movable pins that are outside of this bin. At this point, other placers would check to see if the bounding box of terminals would

Benchmark	# Cells	# Nets	Whitespace		Metal
			easy	hard	layers
ibm01	12028	11753	14.88%	12.00%	4
ibm02	19062	18688	9.58%	4.72%	5
ibm07	44811	44681	10.05%	4.70%	5
ibm08	50672	48230	9.97%	4.84%	5
ibm09	51382	50678	9.76%	4.88%	5
ibm10	66762	64971	9.78%	4.92%	5
ibm11	68046	67422	9.89%	4.67%	5
ibm12	68735	68376	14.78%	9.94%	5

Table 2: Statistics of the IBMv2 benchmarks [25].

contain the centers of the potential child bins (or would be checking for this condition while gathering the terminals on this net) and stop without adding this net to the partitioning problem. If this condition holds, the net is inessential to partitioning when optimizing for HPWL, but may not be inessential when optimizing for Steiner WL. Thus we cannot skip this net before calculating its three costs.

We calculate the three costs for each net by making calls to a particular Steiner evaluator. If the number of unique points that needs to be passed to the Steiner evaluator is larger than a certain threshold, we use HPWL evaluation instead purely for speed concerns. After making calls to the Steiner evaluator, we make checks to ensure consistency of the costs since the evaluators we are using are approximation algorithms for building RSMTs. For example we ensure that $c_1 \leq c_{12}$ by setting $c_1 = min(c_1, c_{12})$ and similarly for c_2. Also, we make sure that c_{12} is no larger than $min(c_1, c_2)+$ the rectilinear distance between the centers of the child bins. This is necessarily true because one has a tree that connects to all the terminals on the net and the center of partition 1, one can easily connect to the center of partition 2 with a single edge.

After constructing the partitioning instance with properly weighted nets, the partitioner runs and produces a solution. A cutline is selected based on the partitioning (see Section 5 for more details), and then new bins are constructed for the next cycle of min-cut placement to continue. When the new bin is constructed, cells that belong to that bin are placed at its center and all pointsets for nets incident to the bin must be updated. Since the pointset structures are sorted and have multiplicities, moving a pin to a new location takes time logarithmic in the number of pins on a net. For a given placement instance, this time is essentially constant. Without multiplicities, the entire pointset would need to be rebuilt from scratch due to the removal of duplicates. Empirically building and maintaining the pointset data structures takes less than 1% of the runtime of global placement. Pointsets must also be updated when when bin is placed — movable pins get reassigned to the fixed-pin pointset. Note that partitioning only causes a movable pin to change position, and fixed pointsets are unaffected.

Performance. After implementing net-weighting based on pointsets, we compared three different Steiner evaluators to see their impact on runtime and solution quality. Based on the results discussed in Appendix B, we have chosen FastSteiner [16] for global placement, thanks to its reasonable runtime and consistent performance on large nets. Table 3 shows that the use of FastSteiner with our techniques lead to a reduction of StWL on IBMv2 benchmarks [25] by nearly 3% on average while using less than 30% additional runtime. Since min-cut placers are fast and extremely scalable, this is a very encouraging result. The largest and smallest benchmarks (ibm01e and ibm12e) differ by 5x in size, but HPWL minimization consistently takes 75% of runtime for StWL minimization, suggesting that the ratio remains approximately constant regardless of the scale.

Benchmark	Minimizing HPWL			Minimizing Steiner WL		
	HPWL	StWL	Time (s)	HPWL	StWL	Time (s)
ibm01e	0.523	0.602	205	0.526	0.590	271
ibm01h	0.514	0.592	204	0.523	0.587	266
ibm08e	3.603	4.300	1484	3.757	4.241	2304
ibm08h	3.620	4.258	1446	3.646	4.101	2268
ibm12e	8.193	9.109	2235	8.321	8.990	3016
ibm12h	7.983	8.907	2215	7.966	8.621	2957
Ratio	0.987	1.029	0.733	1.000	1.000	1.000

Table 3: Improving Steiner WL with FastSteiner [16]. Average HPWL, Steiner WL and placement runtimes are shown for representative IBMv2 benchmarks [25]. All wirelengths are in meters. Optimizing StWL decreases StWL by 2.9%, increases runtime by 27% and increases HPWL by 1.3%.

4. DETAIL PLACEMENT DRIVEN BY STEINER TREE LENGTH

Sliding-window optimizations for HPWL during detail placement are quite common in modern placers. A recent techniques of that variety models single-trunk Steiner trees and has had success in improving routability of FPGAs [13]. Unfortunately, it appears very slow. We have implemented two types of sliding-window optimizers directed at minimizing StWL using the FLUTE Steiner evaluator [8]. The first optimizer checks all possible linear orderings of small groups of cells *exhaustively*. The second optimizer also does linear placement, but uses a *dynamic programming* algorithm for an interleaving optimization similar in spirit to that presented by Jariwala and Lillis [13]. Given k cells, this dynamic programming algorithm explores an n-by-m ($n = k/2$, $m = k - n$) table of partial solutions, and uses FLUTE to evaluate their costs. Since such dynamic programming does not, in general, produce min-StWL configurations, we keep the better of the initial and dynamic programming solutions. Compared to exhaustive search, dynamic programming allows for larger windows with reasonable runtime.

Table 4 evaluates detail placement on the IBMv2 benchmarks, with 4 cells per window during exhaustive enumeration and 8 cells per window during interleaving. Such detail placement alone reduces Steiner WL by 0.69% and routed WL by 0.72% while only consuming 11.83% of the total placement runtime.

5. CONGESTION-BASED CUTLINE SHIFTING

In this section we introduce whitespace allocation based on congestion estimates during min-cut placement. This technique is essential to achieving routability, but in some cases increases routed wirelength, as seen in Figure 2.

Benchmark	Steiner WL improvement	Routed WL improvement	% Total runtime
ibm02e	0.735%	1.344%	10.89%
ibm02h	0.644%	1.078%	11.14%
ibm09e	0.533%	0.890%	13.00%
ibm09h	0.716%	0.282%	13.26%
ibm12e	0.698%	1.019%	11.11%
ibm12h	0.619%	2.318%	11.50%
Average	0.688%	0.717%	11.83%

Table 4: Detail placement improves Steiner WL and routed WL. Average improvements and runtime (as a fraction of total placement time) are shown for representative IBMv2 benchmarks [25].

One of the most important reasons that we use bisection instead of quadrisection is the flexibility that it allows in choosing the cutline of a partitioned bin. Before partitioning we first choose a direction for the cutline which is usually based upon the geometry of the bin. We then choose a tentative cutline in that direction to split the bin roughly in half.

After the partitioner returns a solution, we have the flexibility to keep the cutline as it was chosen before partitioning or to change it to optimize an objective. The WSA [18] technique, applied after placement, geometrically divides the placement area in half and estimates the congestion in both halves of the layout. It then allocates more area to the side with greater routing demand (i.e. shifts the cutline) and proceeds recursively on the two halves of the design. In WSA, cells must be re-placed after the whitespace allocation. We can avoid this re-placement because our cells have not yet been placed and will be taken care of naturally in the min-cut process.

Cutline shifting used to handle congestion necessitates a slicing floorplan. The only work in the literature that describes top-down congestion estimates and uses them in placement assumes a grid structure [2]. Therefore we develop the following technique: before each round of partitioning, we overlay the entire placement region on a grid. We choose the grid such that each placement bin is covered by 2-4 grid cells. We build a congestion map using the last updated locations of all pins. We choose the mapping technique from [24] as it shows good correlation with routed congestion.

When cells are partitioned and their positions are changed, the congestion values for their nets are updated. Before cutline shifting, the routing demands and supplies for either side of the cutline are estimated with the congestion map. Given the bounding box of a region, we estimate its demand and supply by intersecting the bounding box with the grid cells of the congestion map. Grid cells that partially overlap with the given bounding box contribute only a portion of their demand and supply based on the ratio of the area of the overlap to the area of the grid cell. Using the demand and supply estimates, we shift the cutline to equalize the ratio of demand to supply on either side of the cutline.

To show the effectiveness of this dynamic version of WSA, we plot congestion maps of placements of ibm01h produced with and without our technique in Figure 1. The left plot illustrates uniform whitespace allocation and the right plot congestion-driven whitespace allocation. Our whitespace allocation technique reduces the maximum congestion by 50% and the number of overfull global routing cells from 3.95% to 3.18% (as reported by an industry router). We also post-process our placements with WSA and observe mixed results, as discussed below (see Table 7).

6. EXPERIMENTAL RESULTS

To test the quality of placements produced by ROOSTER, we ran it on the IBMv2 suite of benchmarks [25] and routed them using Cadence WarpRoute 2.4.41. All runs of placement and routing were performed on 3.2GHz Intel Pentium 4 processors with 1GB of RAM. Statistics for the IBMv2 benchmarks are shown in Table 2. A comparison of ROOSTER against the best published results for several competitive placers is shown in Table 5. A ratio greater than 1.0 indicates that our results are overall better for routing on this benchmark suite, which is true for all the routed wirelengths and via counts of previously published results.

Most of the placers whose best published results are shown in Table 5 have more recent binaries which we evaluate in Table 6. We ran Dragon 4.0 in fixed-die mode on the IBMv2 benchmarks, but it consistently crashed and we are unable to show results for it. Table 6 shows that the latest version of mPL-R + WSA has slightly worse rWL (0.7%) when compared to ROOSTER and 6.9% higher via count. APlace 2.04 has rWL 3.24% smaller than ours, but 7.32% more vias and violations on 2 of the 16 benchmarks.

Since our cutline shifting for congestion can be viewed as a dynamic version of the WSA post-processing technique, we were interested in seeing how WSA or other detail placement techniques would affect the routability of our placements. Table 7 shows that WSA is able to improve our wirelength by approximately 1.0% with a 0.4% increase in via count. Direct comparisons show that the most improvement is obtained on the ibm01 and ibm02 benchmarks. In contrast, the detail placers of Dragon 4.0 and FengShui 5.1 make the routability of our placements far worse with increases in routed wirelength, via count and violations.

7. CONCLUSIONS AND FURTHER WORK

We have presented techniques which leverage recent advances in RSMT construction [8, 16] to optimize Steiner wirelength in global and detail placement with only a modest increase in runtime, features currently usable only in our placer ROOSTER which is freely available as part of the UMpack (http://vlsicad.eecs.umich.edu/BK/PDtools/). With our whitespace allocation based on congestion maps from [24], ROOSTER outperforms best published results for Dragon, Capo, FengShui, mPL-R/WSA and APlace in terms of routed wirelength by 10.7%, 5.6%, 9.3%, 5.5% and 4.2% respectively. Via counts, especially important at 90nm and below, are improved by 15.6% over mPL-R/WSA and 11.9% over APlace. Further improvements by other researchers in Steiner-tree construction and congestion maps can only make our results better. In particular, if the FLUTE package becomes faster and can process larger nets with high fidelity, the window sizes used by our detail placers can increase.

Properly accounting for obstacles in placement is an area that we believe could benefit significantly from our StWL minimization techniques. An obstacle-aware Steiner evaluator could be used directly in our implementation for nontrivial improvement. In addition to handling blockages, both Steiner-tree evaluators used in ROOSTER (FLUTE [8] and FastSteiner [16]) can be used with arbitrary per unit-costs of horizontal and vertical wires. This may provide a safe means of balancing the demand for horizontal and vertical routing resources (similarly motivated cut-line selection in min-cut placement did not improve results in our experiments).

Our technique may conceivably be extended to improve circuit timing — this mainly requires the ability to estimate the per-net timing differential based on Steiner trees which we already compute. Extensions to optimize timing may require block-based static timing analysis. Even more accessible would be a similar extension to optimize dynamic power. In particular, in designs with multiple clock domains, we could optimize clock trees during global placement by estimating the lengths of bounded-skew clock trees using algorithms such as BST-DME.

Acknowledgements. This work was partially supported by the Gigascale Silicon Research Center (GSRC) and the National Science Foundation (NSF).

8. REFERENCES

[1] A. Agnihotri et al., "Fractional Cut: Improved Recursive Bisection Placement," *ICCAD*, pp. 307-310, 2003.

[2] U. Brenner and A. Rohe, "An Effective Congestion Driven Placement Framework," *ISPD*, pp. 6-11, San Diego, 2002.

[3] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?," *DAC*, pp. 477-482, Los Angeles, June 2000.

[4] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Design and Implementation of Move-based Heuristics for VLSI Hypergraph Partitioning," *ACM J. of Experimental Algorithms*, vol. 5, 2000.

	ROOSTER			mPL-R + WSA [18]			APlace 1.0 /w cong [17]			Capo 9.2 [19]		Dragon 3.01[25]		FengShui 2.6 [1]	
	rWL	#Vias	#Viol.	rWL	#Vias	#Viol.	rWL	#Vias	#Viol.	rWL	#Viol.	rWL	#Viol.	rWL	#Viol.
ibm01e	**0.733**	**122286**	0	0.77	127969	0	0.80	152489	0	0.779	0	0.843	0	time-out	932
ibm01h	**0.746**	**124307**	0	0.75	129648	0	0.75	150947	0	0.773	23	0.917	84	time-out	2698
ibm02e	2.059	**259188**	0	**1.89**	284396	0	2.05	299306	0	2.183	0	2.085	0	2.201	0
ibm02h	2.004	**262900**	0	**1.94**	296290	0	2.14	315786	0	2.080	0	2.216	0	2.277	0
ibm07e	**4.075**	**476814**	0	4.29	548765	0	4.18	559354	0	4.534	0	4.495	0	4.756	77
ibm07h	4.329	**489603**	0	4.43	579157	0	**4.29**	586129	1	4.591	0	4.523	0	4.707	251
ibm08e	**4.242**	**559636**	0	4.58	661733	0	4.58	681884	0	4.553	0	4.601	0	4.458	0
ibm08h	**4.262**	**574593**	0	4.49	684910	0	4.63	699411	0	4.768	0	4.961	0	5.056	52
ibm09e	**3.165**	**466283**	0	3.50	549568	0	-	-	-	3.357	0	3.705	0	3.520	0
ibm09h	**3.187**	**475791**	0	3.65	570032	0	-	-	-	3.336	0	3.494	0	3.395	0
ibm10e	**6.412**	**749731**	0	6.84	873311	0	-	-	-	6.591	0	6.948	0	6.809	0
ibm10h	6.602	**775018**	0	6.76	902026	0	-	-	-	**6.484**	0	6.982	0	6.716	0
ibm11e	**4.698**	**605807**	0	5.16	714824	0	-	-	-	5.039	0	5.371	0	5.301	0
ibm11h	**4.697**	**618173**	0	5.15	745015	0	-	-	-	4.941	0	5.400	0	5.260	0
ibm12e	**9.289**	**918363**	0	10.5	1127925	0	-	-	-	9.895	0	10.459	0	10.147	33
ibm12h	**9.289**	**938971**	0	10.1	1107551	0	-	-	-	10.145	0	9.904	0	time-out	3418
Ratio	1.000	1.000		1.055	1.156		1.042	1.119		1.056		1.107		1.093	

Table 5: A comparison of our work to best published routing results for several placers on the IBMv2 benchmarks [25]. All routed wirelengths (rWL) are in meters. A ratio greater than 1.0 indicates that our results are overall better for routing on this benchmark suite. For all cases, ROOSTER outperforms best published routing results in terms of routed wirelength and via count. Published routing data for APlace 1.0 for ibm09-ibm12 is unavailable. Routing data for Capo 9.2, Dragon 3.01 and FengShui 2.6 were taken from [19] which did not contain via counts. Routing uses a 24-hour time-out. Best legal rWL and via counts are in bold.

	Latest mPL-R + WSA			APlace 2.04 -R 0.5			FengShui 5.1		
	rWL	#Vias	#Viol.	rWL	#Vias	#Viol.	rWL	#Vias	#Viol.
ibm01e	**0.718**	123064	0	0.790	158646	85	0.804	166459	1630
ibm01h	**0.691**	213162	0	0.732	161717	2	0.807	166578	1451
ibm02e	**1.821**	**250527**	0	1.846	254713	0	2.324	383169	726
ibm02h	**1.897**	**260455**	0	1.973	268259	0	2.284	343198	148
ibm07e	4.129	492947	0	**3.975**	500574	0	4.387	591002	137
ibm07h	4.240	516929	0	**4.141**	518089	0	4.632	617327	486
ibm08e	4.372	579926	0	**3.956**	588331	0	5.050	740719	19
ibm08h	4.280	599467	0	**3.960**	595528	0	4.759	725147	16
ibm09e	3.319	488697	0	**3.095**	502455	0	3.462	517701	0
ibm09h	3.454	502742	0	**3.102**	512764	0	3.348	510144	0
ibm10e	6.553	777389	0	**6.178**	782942	0	6.599	807032	0
ibm10h	6.474	799544	0	**6.169**	801605	0	6.661	812593	0
ibm11e	4.917	633640	0	4.755	648044	0	5.419	671225	0
ibm11h	4.912	660985	0	4.818	677455	0	5.452	679690	0
ibm12e	10.185	995921	0	8.599	921454	0	9.829	1172981	6
ibm12h	9.724	976993	0	8.814	961296	0	10.333	1344067	466
Ratio	1.007	1.069		0.968	1.073		1.097	1.230	

Table 6: A comparison of our work to the most recent version of mPL-R + WSA, APlace 2.04 and FengShui 5.1 on the IBMv2 benchmarks [25]. All routed wirelengths (rWL) are in meters. The ratios are with respect to ROOSTER's performance described in Table 5. Note that while APlace 2.04 achieves overall smaller wirelength than our placer, it routes with violations on 2 of the 16 benchmarks. Legal rWL and via counts better than those in Table 5 are highlighted in bold.

[5] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Optimal Partitioners and End-case Placers for Top-down Placement," *IEEE Trans. on CAD*, vol. 19, no. 11, pp. 1304-1314, 2000.

[6] C. C. Chang, J. Cong, M. Romesis and M. Xie, "Optimality and Scalabiity Study of Existing Placement Algorithms," *IEEE Trans. on CAD*, pp. 537-549, April 2004.

[7] T. C. Chen, Y. W. Chang and S. C. Lin, "IMF: Interconnect-Driven Multilevel Floorplanning for Large-Scale Building-Module Designs," *ICCAD*, pp. 159-164, November 2005.

[8] C. C. N. Chu, "FLUTE: Fast Lookup Table Based Wirelength Estimation Technique," *ICCAD*, pp. 696-701, 2004.

[9] A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits," *IEEE Trans. on CAD*, vol. 4, no. 1, pp. 92-98, 1985.

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

[11] M. R. Garey and D. S. Johnson, "The Rectilinear Steiner Problem is NP-Complete," *SIAM Journal of Applied Mathematics*, vol. 32, pp. 826-834, 1977.

[12] D. J.-H. Huang, and A. B. Kahng, "Partitioning-based Standard-cell Global Placement With an Exact Objective," *ISPD*, pp. 18-25, 1997.

[13] D. Jariwala and J. Lillis, "Interactions Between Routing and Detailed Placement," *ICCAD*, pp. 387-393, 2004.

[14] A. B. Kahng and G. Robins, "A New Class of Iterative Steiner Tree Heuristics With Good Performance," *IEEE Trans. on CAD*, vol. 11, no. 7, pp. 893-902, July 1992.

[15] A. B. Kahng and X. Xu, "Accurate Pseudo-constructive Wirelength and Congestion Estimation," *SLIP*, pp. 81-86, 2003.

[16] A. B. Kahng, I.I. Mandoiu and A. Zelikovsky, "Highly Scalable Algorithms for Rectilinear and Octilinear Steiner Trees," *ASPDAC*, pp. 827-833, 2003.

[17] A. B. Kahng and Q. Wang, "Implementation and Extensibility of an Analytic Placer," *IEEE Trans. on CAD*, vol. 25, no. 5, pp. 734-747, May 2005.

[18] C. Li, M. Xie, C. K. Koh, J. Cong and P. H. Madden, "Routability-driven Placement and White Space Allocation," *ICCAD*, pp. 394-401, 2004.

[19] J. A. Roy, S. N. Adya, D. A. Papa and I. L. Markov, "Min-cut Floorplacement," to appear *IEEE Trans. on CAD*, 2006.

	ROOSTER + WSA			ROOSTER + Dragon 4.0 DP			ROOSTER + FengShui 5.1 DP		
	rWL	#Vias	#Viol.	rWL	#Vias	#Viol.	rWL	#Vias	#Viol.
ibm01e	**0.718**	122873	0	0.790	133498	0	0.850	162248	155
ibm01h	**0.725**	**124063**	0	0.800	176562	36	0.858	176585	257
ibm02e	**2.000**	**256155**	0	2.164	278854	0	2.215	347022	129
ibm02h	**1.978**	**262022**	0	2.004	271237	0	2.234	345638	285
ibm07e	**3.953**	**470104**	0	4.175	502808	0	4.498	581269	563
ibm07h	**4.091**	**489067**	0	4.721	593629	76	4.885	617061	870
ibm08e	**4.231**	**559010**	0	4.443	598266	0	4.662	684313	276
ibm08h	**4.240**	577879	0	4.491	619733	0	4.794	714798	768
ibm09e	3.200	473605	0	3.392	502967	0	3.718	573996	583
ibm09h	3.205	480961	0	3.328	511174	0	3.688	587486	630
ibm10e	6.420	755673	0	6.759	798405	0	7.214	905508	229
ibm10h	**6.544**	781897	0	6.523	804478	0	6.943	911878	296
ibm11e	4.746	613437	0	4.879	644060	0	5.308	735762	492
ibm11h	4.716	625654	0	4.830	654948	0	5.288	755418	591
ibm12e	9.333	930397	0	9.427	953405	0	9.888	1087932	10
ibm12h	**9.282**	942551	0	9.260	966280	0	9.786	1102197	312
Ratio	0.990	1.004		1.041	1.089		1.114	1.248	

Table 7: Results when applying various post-processors to our placements for the IBMv2 benchmarks [25]. All routed wirelengths (rWL) are in meters. The ratios are with respect to ROOSTER's performance described in Table 5. While WSA shows improvement on some of our placements, it increases routed wirelength and via counts on the largest benchmarks. The detail placers of Dragon 4.0 and FengShui 5.1 make the routability of our placements far worse with increases in routed wirelength and via count on all benchmarks and the addition of violations. Legal improvements to ROOSTER in rWL and via counts are highlighted in bold.

Bench-mark	#Macros	#Nets	Max Edge Degree	Avg Edge Degree	#Nets with Degree > 3
ami33	33	123	34	3.4797	8
ami49	49	408	24	2.2892	19
n10	10	118	4	2.1017	2
n30	30	349	3	2.0716	0
n50	50	485	4	2.1650	1
n100	100	885	4	2.1164	5
n300	300	1893	6	2.3022	47

Bench-mark	Minimizing HPWL			Minimizing Steiner WL		
	HPWL	StWL	Time	HPWL	StWL	Time
ami33	83267	105857	1.20	83434	103566	35.44
ami49	913680	934291	2.90	932408	951646	13.67
n10	56767	56841	0.12	57169	57277	0.45
n30	172614	172614	1.07	170527	170527	3.78
n50	204061	204100	3.16	207151	207193	9.70
n100	339423	339545	12.76	340396	340502	37.05
n300	764859	766389	122.98	760575	761968	299.32
Ratio	1.000	1.000	1.000	1.004	1.001	4.590

Table 8: Fixed-outline floorplanning to minimize HPWL versus Steiner WL. All StWLs were calculated using the Steiner evaluator FLUTE [8]. All wirelength and runtime measures are averaged over 50 runs. Optimizing for Steiner WL increases runtime by a minimum of 2.43x for `n300` and a maximum of 29.53x for `ami33`.

[20] N. Selvakkumaran, P. Parakh and G. Karypis, "Perimeter-degree: A Priori Metric for Directly Measuring and Homogenizing Interconnection Complexity in Multilevel Placement," *SLIP*, pp. 53-59, 2003.

[21] N. Selvakkumaran and G. Karypis, "Theto - A Fast, Scalable and High-quality Partitioning Driven Placement Tool," Technical report, University of Minnesota, 2004.

[22] P. Suaris and G. Kedem, "Quadrisection: A New Approach to Standard Cell Layout," *ICCAD*, pp. 474-477, 1987.

[23] K. Takahashi et al, "Min-cut Placement with Global Objective Functions for Large Scale Sea-of-gates Arrays," *IEEE Trans. on CAD*, vol. 14, no. 4, pp. 434-446, 1995.

[24] J. Westra, C. Bartels and P. Groeneveld, "Probabilistic Congestion Prediction," *ISPD*, pp. 204-209, 2004.

[25] X. Yang, B. K. Choi, and M. Sarrafzadeh, "Routability Driven White Space Allocation for Fixed-die Standard-cell Placement," *ISPD*, pp. 42-49, 2002.

A. OPTIMIZING STEINER WIRELENGTH IN FLOORPLANNING

In order to show just how expensive Steiner length estimation can be as compared to HPWL computation, we replaced the HPWL subroutine of the fixed-outline annealing-based floorplanner Parquet with FLUTE [8], a very fast Steiner-tree evaluator. The choice of floorplanning for this experiment is explained by its relative simplicity. It also clearly illustrates the impact of optimizing Steiner length on runtime and solution quality in circuit layout.

Table 8 shows netlist statistics for some common floorplanning benchmarks as well as runtimes and wirelengths with and without the use of FLUTE. All runtimes and wirelengths are averages over 50 runs. As is evident from the table, blindly replacing an HPWL evaluator with a Steiner-tree evaluator, even one as fast as FLUTE, can result in a huge increase in runtime when nets have nontrivial pin count. Trivial pincount for any Steiner evaluator is three or fewer since Steiner length is the same as HPWL in such instances. All the nets in the `n30` benchmark have trivial pincount, but we observe a 3.53x increase in runtime. The reason for this runtime increase is that calling a Steiner-tree evaluator requires nontrivial overhead (most notably the removal of duplicate points which requires sorting) as compared to Parquet's HPWL evaluator which is hand-tuned for speed.

The data in the table is also quite striking in that it shows that optimizing for Steiner length was not particularly effective, as Steiner wirelength and HPWL were both increased across all of the benchmarks. This shows that what one may think is an obvious method to reduce Steiner wirelength may not be all that useful. A possible explanation of this strange result is that Steiner WL is not a convex objective. Therefore it may require a longer annealing schedule than a convex objective like HPWL, whereas in our experiments the annealing schedule was fixed.

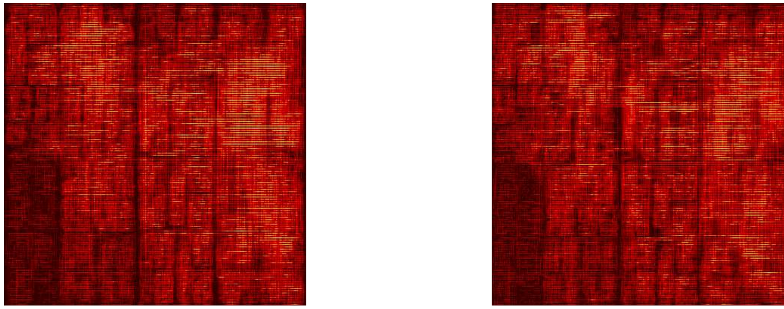

Figure 1: Congestion maps for the ibm01h benchmark: uniform whitespace allocation (produced with Capo -uniformWS) is illustrated on the left, congestion-driven allocation in ROOSTER is illustrated on the right. The peak congestion when using uniform whitespace is 50% greater than that for our technique. When routed with Cadence WarpRoute, uniform whitespace produces 3.95% overfull global routing cells and routes in just over 5 hours with 120 violations. ROOSTER's whitespace allocation produces 3.18% overfull global routing cells and routes in 22 minutes without violations.

B. STEINER-TREE TOOLS: RUNTIME, ACCURACY AND FIDELITY

After implementing our technique to reduce StWL during global placement, we tested three different Steiner-tree evaluators to see how they would affect the runtime and solution quality of placement. The three evaluators used were Batched Iterated 1 Steiner (BI1ST) [14], FastSteiner [16] and FLUTE [8]. We used each evaluator individually as well as combinations of all three. When using more than one evaluator at a time, we choose the smallest wirelength among all estimates since RSMT estimators overestimate actual RSMT length. Recall that FLUTE is known to be optimal for nets with nine or fewer pins and also much faster than other evaluators. Therefore, in mixed evaluators for nets with four to nine pins we use FLUTE exclusively.

Table 9 shows a runtime and solution quality comparison for all eight possible combinations of Steiner evaluator for the benchmark ibm01e. Runtimes and wirelengths are averages of five independent runs. The trends present for ibm01e are very similar for the other IBMv2 benchmarks. It is clear from the table that BI1ST gives the best solutions but uses the most runtime for a single evaluator. FastSteiner is very close to BI1ST in terms of solution quality, but uses much less runtime. Of the three pure evaluators, FLUTE is the least successful in terms of placement quality but is the fastest. We decided to use FastSteiner in global placement because it provided the best trade-off in terms of solution quality and runtime across all benchmarks.

Surprisingly, the mixed Steiner evaluators were outperformed by individual evaluators and hurt solution quality rather than improved it. This trend was even stronger on larger benchmarks. In particular, FastSteiner performed better than FastSteiner + FLUTE on ibm07. Certainly using the best of three Steiner evaluators makes estimates more accurate, but our global placement relies on *differences* between Steiner lengths rather than the *lengths themselves*. This suggests that the accuracy, measured by maximum error, of Steiner-tree estimation is not as important as its *fidelity*, which is defined as preserving *relative* magnitudes between estimates.

Steiner evaluator(s)	Place time (s)	Steiner WL	Steiner WL Ratio
HPWL (no Steiner eval)	**141**	**0.5955**	**1.0000**
BI1ST + FastSteiner + FLUTE	202	0.5918	0.9937
BI1ST + FLUTE	186	0.5900	0.9907
BI1ST + FastSteiner	248	0.5893	0.9895
FLUTE	148	0.5886	0.9884
FLUTE + FastSteiner	158	0.5875	0.9866
FastSteiner	**180**	**0.5875**	**0.9866**
BI1ST	208	0.5861	0.9843

Table 9: Impact of Steiner evaluators during global placement (ibm01e). Total StWL and global placement runtime are listed for all combinations of three Steiner evaluators. In such combinations, the minimum Steiner length estimate is used in weighted partitioning (see details in Appendix B).

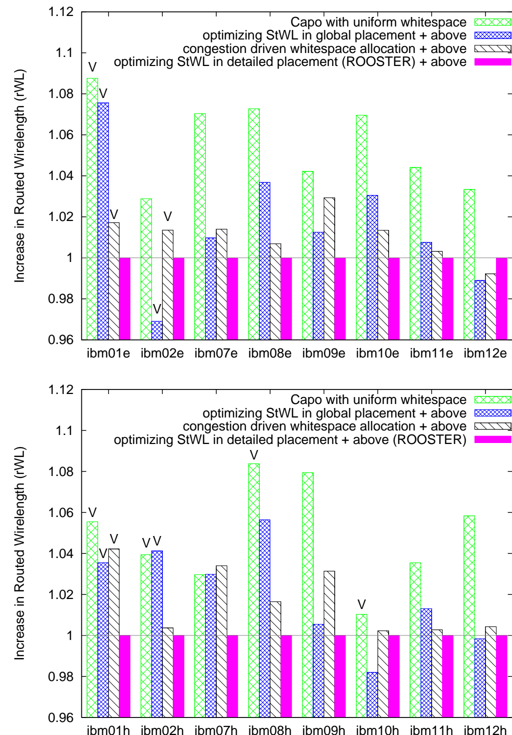

Figure 2: Impact of individual optimizations on the rWL produced by ROOSTER. "V" indicates violations in routing.