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

Jarrod A. Roy and Igor L. Markov
The University of Michigan, Department of EECS
2260 Hayward Ave., Ann Arbor, MI 48109-2121
{royj,imarkov}@eecs.umich.edu

***Abstract*— We demonstrate that Steiner-tree Wirelength (StWL) correlates with Routed Wirelength (rWL) much better than the more common Half-Perimeter Wirelength (HPWL) objective. Therefore, we develop a technique to optimize StWL in global and detail placement without a significant runtime penalty. This new optimization, along with congestion-driven whitespace distribution, improves overall Place-and-Route results, making the use of HPWL unnecessary. Additionally, our empirical results provide ample evidence that the fidelity of net length estimates is more important than their accuracy in Place-and-Route. The new data structures that make our min-cut algorithms fast can also be useful in multi-level analytical placement.**

Our placement algorithm 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.

	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 I
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.

I. INTRODUCTION

Recently there has been much interest in estimating the amount of improvement that is left in placement optimization [9]. 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 — *minimizing wrong objective functions*, 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 Steiner trees.

Our main contribution is a series of optimization techniques for 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, particularly the terminal propagation technique from [27], improved in [10], 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 [16] cannot capture Steiner-tree lengths in bisection or quadrisection (for more details see Section II-A). There has also been work in weighting the HPWL of individual nets based on their pin counts [11]. Later work improved on these weighting techniques [4]. The authors of [5] find that these weighted wirelength techniques are reasonable predictors of routed wirelength, but that smaller weighted wirelength can translate into larger routed wirelength making the use of weighted wirelength as an optimization "questionable."

Our Steiner-tree driven detail placer leverages the speed of the recent FLUTE package [12]. The closest work in detail placement [18] models single-trunk Steiner trees to reduce congestion in FPGAs. While effective, this technique requires exorbitant amounts of runtime. Instead, our detail placer considers optimal Steiner trees and is nevertheless quite fast.

We also build upon recent work in congestion-driven placement that uses congestion maps. In [31], congestion maps are built after global placement, and annealing moves are applied to minimize a congestion metric. Another technique, known as WSA [23], is applied after detail placement. It identifies areas with

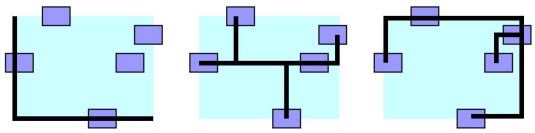


Fig. 1. HPWL (left), Steiner WL (center) and Rectilinear Minimal Spanning Tree (MST) WL (right) for a 5-pin net.

high congestion and injects whitespace into these areas in a top-down fashion. Our work uses congestion maps from [30] 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 [31].

At the 90nm technology node and below, increased via resistance, manufacturing variability and manufacturing defects require unprecedented attention to vias. In particular, via resistance may vary by more than other important circuit parameters — in some technologies a difference of 30 times has been observed between neighboring vias. Therefore, manufacturers prefer and sometimes require vias to be doubled, since this averages out the variation. 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.

In the remainder of this paper, Section II describes previous work on VLSI placement. Section III discusses choosing the right objective to optimize in placement and outlines a first implementation in floorplanning. Sections IV and V introduce the realization of Steiner-tree modeling in min-cut placers and Steiner-driven detail placement, respectively. Section VI outlines whitespace allocation to improve routability. Experimental results are given in Section VII, and Section VIII concludes and motivates further applications of our techniques.

II. 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. Figure 1 shows how HPWL, RSMT, and

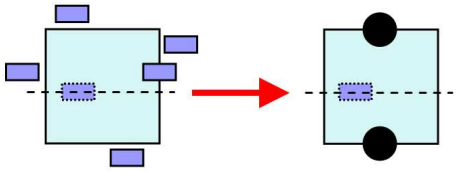


Fig. 2. Propagation of the terminals of a net to the sides of the bin above and below the proposed cutline. The net has five fixed terminals: four above and one below the cutline. The net also has movable cells which are represented by the cell with a dashed outline. The four fixed terminals above the cutline are propagated to the black circle at the top of the bin while the one fixed terminal below the cutline is propagated to the black circle below the cutline. The movable cells remain unpropagated. Note that the net is inessential since terminals are propagated to both sides of the cutline.

Minimal Spanning Tree (MST) length differ for a given 5-pin net. Note that the shortest vertical segment in the RSMT is not included in the HPWL of the net. Since RSMT construction is an NP-complete problem [15], it has been generally regarded as too computationally demanding for use in placement [16]. To illustrate how a placer optimizes its chosen objective, we describe a specific technique – top-down min-cut placement.

A. Top-down Min-cut 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 [5]. 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 [6], [14], or alternatively with quadratic placement and geometric partitioning [3].

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 [29].

Terminal propagation and inessential nets. Proper handling of terminals is essential to the success of top-down placement approaches [7], [13], [16], [28]. 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 [7]. Terminal propagation is typically driven by geometric proximity of terminals to subregions/partitions. Figure 2 depicts terminal propagation for a net with several fixed terminals. This particular net is inessential as it has terminals propagated to both sides of the cutline.

Minimizing HPWL through weighted net-cut. The authors of [27] 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 [27], 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 [10] and reduced to the calculation of three wirelengths per net per partitioning instance (see more details in Section IV). 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 [16], 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 in order for a wirelength evaluator to be feasible for placement optimization, it must have evaluation complexity equal to or lesser than MST. On the other hand, the authors claim that their techniques can apply to "arbitrarily complicated per-net placement objectives" [16].

The net-vector technique includes the computation of 2^p integer costs per optimization objective defined for p partitions ($p = 4$ in [16] 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 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.

B. 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 [22], [30]. In this work, we make use of the congestion mapping techniques presented in [30] which assumes that routers attempt to route nets with the fewest number of bends possible. The technique 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 [30] 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 [30].

Rectilinear Steiner Minimal tree evaluators. The problem of constructing Rectilinear Steiner Minimal trees is known to be NP-hard [15]. 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) [20], Fast-Steiner [19], and FLUTE [12]. BI1ST, albeit the oldest

and slowest of these algorithms, generally produces the best solutions overall. FLUTE, the most recent and fastest algorithm, is provably optimal for instances with nine points or fewer. FastSteiner falls in the middle in terms of both speed and solution quality.

C. 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 [31], congestion maps are built after global placement, and annealing moves are applied to minimize a congestion metric. Another technique known as WSA [23] 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 whitespace allocation, cells typically 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 [26] and cell spreading [23] are used to tie whitespace to specific cells, rather than to fixed regions as in techniques based on congestion maps.[1]

III. Choosing the Proper Objective

In this section we seek a wirelength estimator that adequately captures routed wirelength and is suitable for efficient optimization. While the former appears within reach, the latter turns out more difficult.

A. Estimating Net Length

A priori wirelength estimation is the subject of extensive literature [4]. In this work we are mainly interested in evaluating and using simple per-net estimators, such as weighted HPWL, identified previously as a reasonable compromise between HPWL and Rectilinear Steiner Minimal Tree (RSMT) evaluators [4]. However, experiments described in [5] reveal poor correlation between total weighted HPWL and total routed WL in placement. Therefore, we do not consider weighted HPWL as a potential objective in our work.

On the positive side, recent progress on fast RSMT evaluators [12], [19] opens the possibility of using them in optimization. HPWL and RSMT WL (aka Steiner WL) share the same drawback — they both underestimate routed wirelength (rWL), due to detours, pin access problems, etc. A common response to this issue is to use the Minimal Spanning Tree length (MSTWL) [16]; this is relatively easy to compute and does not exceed Steiner WL by more than 50%. Therefore, we also include MSTWL in our experiments.

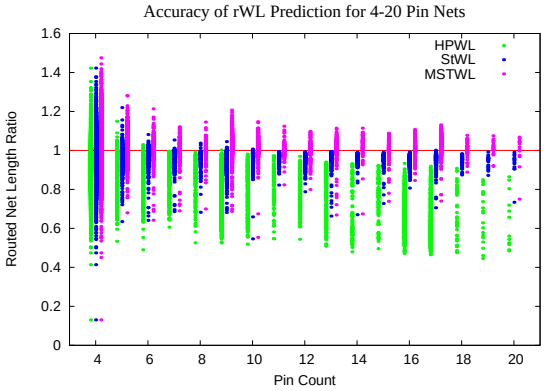


Fig. 3. Comparing the accuracy of routed wirelength (rWL) estimators HPWL (left lines), StWL (middle) and MST WL (right) for nets with 4-20 pins in the `vga_lcd` design from the IWLS 2005 benchmarks [17]. StWL was calculated using FastSteiner [19].

To test our intuition, we perform the following experiment. We analyze a placement of the `vga_lcd` design from the IWLS 2005 series of benchmarks [17] which was routed without violation by Cadence WarpRoute. The `vga_lcd` design has 124,031 standard cells and 124,098 nets. For each net with 4-20 pins, we plot the ratios of HPWL, StWL and MSTWL (length of the MST of the net) to routed net length vs. the pin count of the net. See Figure 1 for a comparison of HPWL, StWL and MSTWL for a 5-pin net. StWL was calculated using FastSteiner [19]. Statistics for 2- and 3-pin nets are not shown as HPWL and StWL produce identical numbers. For each net, three values are plotted in Figure 3: $\frac{HPWL}{rWL}$ (in green, left), $\frac{StWL}{rWL}$ (in blue, middle) and $\frac{MSTWL}{rWL}$ (in purple, right). Nets are separated by their pin counts. In some cases, HPWL and StWL have ratios greater than 1.0. This is due to routers making use of internal wiring within cells that does not count toward reported wirelength. The discrepancy is exacerbated by wide pins present in many cell libraries, as well as by logically and electrically equivalent pins.

Figure 3 shows that HPWL is a poor estimator of routed net length — it can significantly under-estimate rWL and includes a great amount of noise since the range of ratios to rWL is large. As one might expect, StWL typically underestimates routed net length as well, but its range of ratios in the figure is significantly smaller than for MST. This means that with a proper correction, StWL may be a more accurate estimate than MST. [2] More importantly, given two nets, StWL

[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.

[2] Figure 3 suggests that MST is the most accurate estimator of routed net length *on average* for the router used on this design because the ranges of ratios for MST are centered at 1.0.

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 (s)	HPWL	StWL	Time (s)
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 II

FIXED-OUTLINE FLOORPLANNING TO MINIMIZE HPWL VERSUS STEINER WL. ALL STWLS WERE CALCULATED USING THE STEINER EVALUATOR FLUTE [12]. ALL WIRELENGTH AND RUNTIME MEASURES ARE AVERAGED OVER 50 RUNS. OPTIMIZING STEINER WL INCREASES RUNTIME BY A MINIMUM OF 2.43X FOR `n300` AND A MAXIMUM OF 29.53X FOR `ami33`.

estimates can predict more reliably which net will have longer routed length, i.e., StWL has higher *fidelity*. Further experiments described in the Appendix have shown that the fidelity of net length estimates, rather than their accuracy is key in placement. Indeed we have independently verified using MSTWL as an optimization objective is worse than StWL for routability and may be less effective than HPWL in certain situations (see Table XII and discussion in Section VII).

B. Impact of Steiner-tree Evaluation

As a first attempt at optimizing Steiner WL, we replaced the HPWL subroutine of the fixed-outline annealing-based floorplanner Parquet with FLUTE [12], 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 II shows the 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 pin count 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 pin count, 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 [8].

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. One possible explanation of this strange result is that Steiner WL is not a convex objective. Thus, it may require a longer annealing schedule than a convex objective like HPWL, whereas in our experiments the annealing schedule was fixed.

Our empirical results suggest that Simulated Annealing is not compatible with Steiner WL evaluation as Simulated Annealing relies on frequent net length computation, making Steiner WL calculation the bottleneck. Furthermore, Simulated Annealing appears to be ineffective in optimizing Steiner WL as Steiner WL increased on average in our experiments. We pursue a different approach and, surprisingly, manage to optimize Steiner WL with only a modest runtime penalty.

IV. 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 7.

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 [27] with a 7-case analysis. A different group reduced this technique to the calculation of three wirelengths per net when building a partitioning instance and verified resulting empirical improvements [10]. To be clear, the three wirelengths that must be calculated per net (w_1, w_2 and w_{12}) completely determine the connectivity and costs of all nets in the derived partitioning hypergraph [10].

While the formulation from [10] is more compact than the one from [27], we also note that it is far more general. For each net in a partitioning instance, one must calculate the cost of all nodes on the net being placed at the center of partition 1 (w_1), the cost of all nodes on the net being placed at the center of partition 2 (w_2) and the cost of all nodes on the net being split between the centers of partitions 1 and 2 (w_{12}).

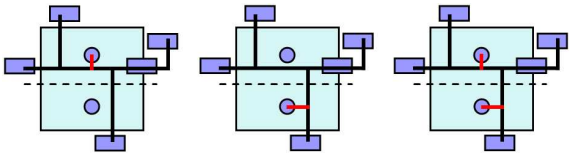


Fig. 4. Calculating the three costs for weighted terminal propagation with StWL: w_1 (left), w_2 (middle), and w_{12} (right). The net has five fixed terminals: four above and one below the proposed cutline. For the traditional HPWL objective, this net would be considered inessential. Note that the structure of the three Steiner trees may be entirely different, which is why w_1, w_2 and w_{12} must be evaluated independently.

For each net of the netlist hypergraph relevant to the partitioning instance, two nets are created in the partitioning hypergraph: one with weight $w_{12} - max(w_1, w_2)$ $|w_1 - w_2|$ and the other with weight $|w_1 - w_2|$ [10]. The new net with weight $w_{12} - max(w_1, w_2)$ connects all of the movable objects (non-terminals) of the original net. The new net with weight $|w_1 - w_2|$ connects all of the movable objects of the original net to one of the fixed terminals in either partition 1 or 2. This new net connects to the terminal in partition 2 when $w_1 > w_2$ and to the terminal in partition 1 when $w_1 < w_2$. If either net has weight 0, it is discarded from the problem. The authors of [10] show, assuming $w_{12} \geq max(w_1, w_2)$, that this net weighting scheme ties minimizing HPWL directly to minimizing the weighted net-cut of the partitioning hypergraph.

The points required to calculate w_1 for a net are the positions of the terminals on the net plus the center of partition 1. Similarly, the points required to calculate w_2 are the positions of the terminals plus the center of partition 2. Lastly, the points to calculate w_{12} are the positions of the terminals on the net plus the centers of both partitions. See Figure 4 for an example of cost calculation. Clearly, the StWL 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. Since StWL satisfies the assumptions made by the authors of [10], weighted partitioning can be used to minimize StWL. 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 [7] — standard techniques for HPWL minimization — cannot be used when minimizing StWL. Moving terminal locations drastically impacts Steiner-tree topology and can make StWL estimates poor. Nets that are considered inessential in HPWL minimization are not necessarily inessential when considering StWL because there are many Steiner trees of different lengths that have the same bounding box. Figure 4 illustrates a net that is inessential for HPWL minimization but essential for StWL minimization.

Pointsets with multiplicities. Building Steiner trees for each net during partitioning is a computationally expensive task. Table II in Section III-B shows how expensive a naive replacement of HPWL with Steiner-tree evaluation can be in floorplanning. Even traversing nets to collect all relevant point locations when building Steiner trees 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.

To keep 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 represent terminals, and fixed and placed 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 corresponds to a given point. Before legalization in detail placement, cell overlap can cause pins to have the same location.

Maintaining the number of real pins that corresponds to a point in a pointset (i.e., the multiplicity of that point) is necessary for efficient update of pin locations during placement. If a pin changes position during placement, the pointsets for the net connected to the pin must be updated. First, the original position of the pin must be removed from the movable point set. To remove the pin, one performs a binary search on the pointset. As multiple pins can have the same position, especially early in placement, without pointset the entire net would need to be traversed to see if any other pins share the same position as the pin that is moving. However, multiplicities make this information available in constant time. After the pin's location is found in the pointset, its multiplicity is reduced by 1. If this results in the position having a multiplicity of 0, the position is removed entirely. Insertion of the pin's new position is similar: first, a binary search is performed on the pointset. If the position is present, it's multiplicity is increased by 1. Otherwise, the position is added in sorted order with multiplicity 1.

Steiner weighted min-cut step by step. Pseudocode for minimizing Steiner wirelength in global placement is illustrated in Figure 5. At the beginning of min-cut placement, all 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. To initialize a pointset, we sort it and change duplicates to multiplicities in a linear-time pass.

```
Variables: queue of placement bins
Initialize queue with top-level placement bin
Initialize pointsets with all movable pins at the
  center of the top-level placement bin and
  fixed pins at their fixed locations
 1 While (queue not empty)
 2   Dequeue a bin
 3   If (bin small enough)
 4     Process bin with end-case placer
 5     Update pointsets for all nets affected by
         cell placement (make movable pins fixed)
 6   Else
 7     Choose a cut-line for the bin
 8     Calculate the centers of the child bins
 9     Build a partitioning hypergraph from netlist
         and cells contained in the bin
10       Foreach (net adjacent to a cell in the bin)
11         Build a list of terminal pin locations on
             the net by combining all points from the
             net's fixed pointset and points from the
             net's movable pointset outside the bin
12         Calculate w1 from terminal locations
             and center of child bin 1 using
             Steiner evaluator(s)
13         Calculate w2 from terminal locations
             and center of child bin 2 using
             Steiner evaluator(s)
14         Calculate w12 from terminal locations
             and centers of child bins 1 and 2
             using Steiner evaluator(s)
15         Adjust w1,w2,and w12 for consistency
16         Add two nets to partitioning hypergraph
             whose weights and connectivity are
             determined by w1,w2,and w12
17     Bisect the bin into two child bins
18     Update pointsets for all nets affected
         by cell movement
19     Enqueue each child bin
```

Fig. 5. Minimizing StWL in top-down min-cut global placement.

Global Placement Task	Runtime
Partitioning	53.56%
Partitioning problem construction	29.50%
End-case Placement	7.77%
Congestion Maps	6.44%
Pointset Maintenance	**0.86%**
Miscellaneous	1.87%

TABLE III

RUNTIME BREAKDOWN OF GLOBAL PLACEMENT WHEN MINIMIZING STWL FOR IBM01-EASY OF THE IBMV2 SERIES OF BENCHMARKS [31]. "PARTITIONING PROBLEM CONSTRUCTION" INCLUDES RUNTIME FOR STEINER WL EVALUATORS.

Bench-mark	# Cells	# Nets	Whitespace		Metal layers
			easy	hard	
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 IV

STATISTICS OF THE IBMV2 BENCHMARKS [31].

Before a partitioning instance is 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 step, other placers would check to see if the bounding box of terminals would 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. MST WL can be used for these large nets, but we have found routed wirelength degradation as compared to using HPWL (see Table XII). 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 $w_1 \leq w_{12}$ by setting $w_1 = min(w_1, w_{12})$ and similarly for w_2. Also, we make sure that w_{12} is no larger than $min(w_1, w_2)+$ 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 VI for more details), and new bins are constructed for the next cycle of min-cut placement to continue. When a 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. 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, shown in Table III. Pointsets must also be updated 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

Bench-mark	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
ibm02e	1.487	1.745	483	1.526	1.716	738
ibm02h	1.441	1.694	470	1.471	1.654	725
ibm07e	3.482	3.854	1134	3.484	3.747	1480
ibm07h	3.322	3.682	1092	3.401	3.659	1444
ibm08e	3.630	4.300	1484	3.757	4.241	2304
ibm08h	3.608	4.258	1446	3.646	4.131	2268
ibm09e	3.065	3.465	1207	3.130	3.408	1599
ibm09h	2.991	3.390	1179	3.037	3.313	1565
ibm10e	6.016	6.736	1918	6.088	6.619	2541
ibm10h	5.826	6.542	1885	5.830	6.356	2519
ibm11e	4.591	5.003	1740	4.608	4.888	2109
ibm11h	4.430	4.843	1679	4.478	4.757	2064
ibm12e	8.193	9.109	2235	8.321	8.990	3016
ibm12h	7.983	8.907	2215	7.966	8.621	2957
Ratio	1.000	1.000	1.000	1.014	0.972	1.364

TABLE V

IMPROVING STEINER WL WITH FASTSTEINER [19]. AVERAGE HPWL, STEINER WL AND PLACEMENT RUNTIMES ARE SHOWN FOR THE IBMV2 BENCHMARKS [31]. RESULTS ARE THE AVERAGE OF FIVE INDEPENDENT RUNS. ALL WIRELENGTHS ARE IN METERS. OPTIMIZING STWL DECREASES STWL BY 2.8%, INCREASES RUNTIME BY 36% AND INCREASES HPWL BY 1.4%.

based on pointsets, we compared three different Steiner evaluators to see their impact on runtime and solution quality. Based on the results discussed in the Appendix, we have chosen FastSteiner [19] for global placement, due to its reasonable runtime and consistent performance on large nets. Table V shows that the use of FastSteiner with our techniques lead to a reduction of StWL on IBMv2 benchmarks [31] by nearly 3% on average while using 36% 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.

V. DETAIL PLACEMENT DRIVEN BY STEINER TREE LENGTH

Sliding-window optimizations for HPWL during detail placement are quite common in modern placers. A recent technique of that variety models single-trunk Steiner trees and has had success in improving routability of FPGAs [18]. Unfortunately, it appears very slow. We have implemented two types of sliding-window optimizers directed at minimizing StWL using the FLUTE Steiner evaluator [12]. The first optimizer checks all possible linear orderings of small groups of cells and pieces of whitespace *exhaustively*. For the sake of efficiency, orderings of cells that are the same except for permutations of whitespace pieces are only evaluated once. Other than this simple optimization, every cell ordering is generated and its StWL is calculated using FLUTE. The ordering with the least StWL is returned at the end of the procedure. Because of the exponential rate of growth of the number of permutations of n cells, namely $n!$, this exhaustive enumeration technique only scales to 4-5 cells.

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 [18]. Given k cells, the algorithm splits the cells into groups A and B of sizes $n = k/2$ and $m = k - n$, respectively. The order of the cells in groups A and B is important and is the same as the initial configuration to the optimizer. The configurations that the algorithm examines are only those where cells in groups A and B are interleaved, but the relative order of cells from A and cells from B remain unchanged. For example, say we have the cells 1234abcd in this order. The ordering "1ab2cd34" is a legal ordering for the algorithm to consider, but the ordering "12a3bdc4" is not because c came before d in group B previously, but c is now behind d. The exact number of configurations that satisfy this interleaved ordering is $\frac{(n+m)!}{n!m!}$ which is much less than the $(n+m)! = k!$ possible configurations of the input.

First, the algorithm builds an n-by-m sized table of partial solutions. Entry (i, j) of the table contains the ordering with the best (smallest) StWL when interleaving the first i elements of group A and the first j elements of group B. The final answer is thus stored in position (n,m) of the table after the algorithm finishes. Table entries $(i,0)$ and $(0, j)$ are trivial to calculate. The dynamic programming step of the algorithm computes entry (i, j) from entries $(i-1, j)$ and $(i, j-1)$. Element i of group A is added to the solution from entry $(i-1, j)$ and the StWL of the resulting placement is calculated from scratch with FLUTE. Similarly, element j of group B is added to the solution from entry $(i, j-1)$ and the StWL of this placement is calculated from scratch with FLUTE. The best of these two solutions in terms of StWL is taken to be the solution for entry (i, j). Calculating entries in row-major (or column-major) order will guarantee that all table entry dependencies are satisfied.

Since the algorithm proceeds by filling in the table, the runtime of the algorithm is proportional to $n * m$ multiplied by the time to evaluate wirelength, while considering $\frac{(n+m)!}{n!m!}$ configurations. To speed up the process of evaluating wirelength, pointsets with multiplicities (see Section IV) are used in interleaving as well as exhaustive search. This dynamic programming approach has been shown to produce the optimal interleaving when HPWL is used for evaluation [18], but we have found that it does not necessarily produce min-StWL interleavings. On the other hand, it allows for windows of size 8-9 which is nearly twice that of exhaustive search.

Table VI evaluates detail placement on the IBMv2

Bench-mark	Steiner WL improvement	Routed WL improvement	% Total runtime
ibm01e	1.047%	1.668%	11.66%
ibm01h	0.950%	4.046%	11.99%
ibm02e	0.735%	1.332%	10.89%
ibm02h	0.644%	0.363%	11.14%
ibm07e	0.647%	1.377%	11.51%
ibm07h	0.622%	3.288%	11.92%
ibm08e	0.553%	0.680%	11.27%
ibm08h	0.540%	1.620%	11.77%
ibm09e	0.716%	2.846%	13.00%
ibm09h	0.698%	3.041%	13.26%
ibm10e	0.662%	1.327%	12.42%
ibm10h	0.642%	0.225%	12.70%
ibm11e	0.639%	0.313%	11.65%
ibm11h	0.607%	0.273%	11.82%
ibm12e	0.682%	-0.789%	11.11%
ibm12h	0.619%	0.423%	11.50%
Average	0.688%	1.387%	11.83%

TABLE VI

DETAIL PLACEMENT IMPROVES STEINER WL AND ROUTED WL. AVERAGE IMPROVEMENTS AND RUNTIME (AS A FRACTION OF TOTAL PLACEMENT TIME) ARE SHOWN FOR THE IBMV2 BENCHMARKS [31]. RESULTS ARE THE AVERAGE OF FIVE INDEPENDENT RUNS.

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 1.4% while only consuming 11.8% of the total placement runtime.

VI. 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 7.

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, 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 [23] 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. However, we can avoid this re-placement because our cells have not yet been placed and will be taken care of naturally during 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 [3]. 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 then build a congestion map using the last updated locations of all pins. We choose the mapping technique from [30] 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 these, 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 8. 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 IX).

VII. EXPERIMENTAL RESULTS

To test the quality of placements produced by ROOSTER, we ran it on the IBMv2 suite of benchmarks [31] 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. All runs of randomized placers, including ROOSTER, are the average results for the best of three independent placements (only the best of the three independent placements is routed and the results of three such sets of placements are averaged). Statistics for the IBMv2 benchmarks are shown in Table IV. The effectiveness of each of the approaches that make up ROOSTER is depicted in Figure 7. A comparison of ROOSTER against the best published results for several competitive placers is shown in Table VII. 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 VII have more recent binaries which we evaluate in Table VIII. We ran Dragon 4.0

	ROOSTER			mPL-R + WSA [23]			APlace 1.0 /w cong [21]			Capo 9.2 [24]		Dragon 3.01 [31]		FengShui 2.6 [2]	
	rWL	#Vias	#Vio.	rWL	#Vias	#Vio.	rWL	#Vias	#Vio.	rWL	#Vio.	rWL	#Vio.	rWL	#Vio.
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 VII

A COMPARISON OF OUR WORK TO BEST PUBLISHED ROUTING RESULTS FOR SEVERAL PLACERS ON THE IBMV2 BENCHMARKS [31]. 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 [24] WHICH DID NOT CONTAIN VIA COUNTS. ROUTING USES A 24-HOUR TIME-OUT. BEST LEGAL RWL AND VIA COUNTS ARE IN BOLD.

	ROOSTER				Latest mPL-R + WSA				APlace 2.04 -R 0.5				FengShui 5.1			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
ibm01e	0.733	**122286**	0	42	**0.718**	123064	0	11	0.790	158646	85	132	0.804	166459	1630	1337
ibm01h	0.746	**124307**	0	32	**0.691**	213162	0	11	0.732	161717	2	121	0.807	166578	1451	1310
ibm02e	2.059	259188	0	13	**1.821**	**250527**	0	11	1.846	254713	0	9	2.324	383169	726	474
ibm02h	2.004	262900	0	14	**1.897**	**260455**	0	13	1.973	268259	0	14	2.284	343198	148	184
ibm07e	4.075	**476814**	0	17	4.130	492947	0	21	**3.975**	500574	0	17	4.387	591002	137	84
ibm07h	4.329	**489603**	0	19	4.240	516929	0	26	**4.141**	518089	0	23	4.632	617327	486	244
ibm08e	4.242	**559636**	0	17	4.372	579926	0	23	**3.956**	588331	0	18	5.050	740719	19	112
ibm08h	4.262	**574593**	0	20	4.280	599467	0	26	**3.960**	595528	0	18	4.759	725147	16	59
ibm09e	3.165	**466283**	0	11	3.319	488697	0	17	**3.095**	502455	0	11	3.462	517701	0	13
ibm09h	3.187	**475791**	0	11	3.454	502742	0	19	**3.102**	512764	0	12	3.348	510144	0	13
ibm10e	6.412	**749731**	0	22	6.553	777389	0	30	**6.178**	782942	0	23	6.599	807032	0	24
ibm10h	6.602	**775018**	0	27	6.474	799544	0	33	**6.169**	801605	0	28	6.661	812593	0	27
ibm11e	**4.698**	**605807**	0	15	4.917	633640	0	22	4.755	648044	0	18	5.419	671225	0	22
ibm11h	**4.697**	**618173**	0	16	4.912	660985	0	25	4.818	677455	0	24	5.452	679690	0	22
ibm12e	9.289	**918363**	0	36	10.185	995921	0	57	**8.599**	921454	0	32	9.829	1172981	6	73
ibm12h	9.289	**938971**	0	43	9.724	976993	0	50	**8.814**	961296	0	50	10.333	1344067	466	448
Ratio	1.000	1.000			1.007	1.069			0.968	1.073			1.097	1.230		

TABLE VIII

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 [31]. ALL ROUTED WIRELENGTHS (RWL) ARE IN METERS. "TIME" REPRESENTS ROUTING RUNTIME IN MINUTES. NOTE THAT WHILE APLACE 2.04 ACHIEVES OVERALL SMALLER WIRELENGTH THAN OUR PLACER, IT ROUTES WITH VIOLATIONS ON 2 OF THE 16 BENCHMARKS. BEST LEGAL RWL AND VIA COUNTS ARE IN BOLD.

in fixed-die mode, but it consistently crashed and we are unable to show results for it. Table VIII 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. Congestion-driven APlace 2.04 (using congestion parameter 0.5) 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 IX 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.

The Faraday series of five mixed-size benchmarks with routing information is derived from circuits released by the Faraday Corporation [1]. To see if ROOSTER techniques are applicable when fixed obstacles are present, we fixed the movable macros in the design (as shown in Figure 6) and used the resulting benchmarks with ROOSTER. All benchmarks were routed using Cadence WarpRoute 2.4.41. A comparison of ROOSTER placements to the original placements of the benchmarks produced by Silicon Ensemble Ultra v5.4.126 (details on the construction

	ROOSTER				ROOSTER + WSA				ROOSTER + Dragon 4.0 DP				ROOSTER + FengShui 5.1 DP			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
ibm01e	0.733	**122286**	0	42	**0.718**	122873	0	7	0.790	133498	0	92	0.850	162248	155	152
ibm01h	0.746	124307	0	32	**0.725**	**124063**	0	10	0.800	176562	36	166	0.858	176585	257	265
ibm02e	2.059	259188	0	13	**2.000**	**256155**	0	10	2.164	278854	0	19	2.215	347022	129	77
ibm02h	2.004	262900	0	14	**1.978**	**262022**	0	11	2.004	271237	0	33	2.234	345638	285	171
ibm07e	4.075	476814	0	17	**3.953**	**470104**	0	13	4.175	502808	0	19	4.498	581269	563	44
ibm07h	4.329	489603	0	19	**4.091**	**489067**	0	19	4.721	593629	76	21	4.885	617061	870	86
ibm08e	4.242	559636	0	17	**4.231**	**559010**	0	16	4.443	598266	0	18	4.662	684313	276	27
ibm08h	4.262	**574593**	0	20	**4.240**	577879	0	19	4.491	619733	0	36	4.794	714798	768	207
ibm09e	**3.165**	**466283**	0	11	3.200	473605	0	11	3.392	502967	0	11	3.718	573996	583	22
ibm09h	**3.187**	**475791**	0	11	3.205	480961	0	11	3.328	511174	0	12	3.688	587486	630	19
ibm10e	**6.412**	**749731**	0	22	6.420	755673	0	21	6.759	798405	0	23	7.214	905508	229	18
ibm10h	6.602	**775018**	0	27	**6.544**	781897	0	26	6.523	804478	0	29	6.943	911878	296	34
ibm11e	**4.698**	**605807**	0	15	4.746	613437	0	15	4.879	644060	0	15	5.308	735762	492	59
ibm11h	**4.697**	**618173**	0	16	4.716	625654	0	16	4.830	654948	0	16	5.288	755418	591	77
ibm12e	**9.289**	**918363**	0	36	9.333	930397	0	30	9.427	953405	0	39	9.888	1087932	10	44
ibm12h	9.289	**938971**	0	43	**9.282**	942551	0	39	9.260	966280	0	47	9.786	1102197	312	66
Ratio	1.000	1.000			0.990	1.004			1.041	1.089			1.114	1.248		

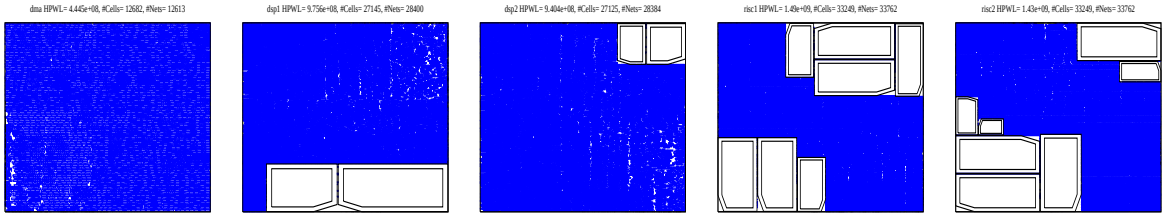
TABLE IX

RESULTS WHEN APPLYING VARIOUS POST-PROCESSORS TO OUR PLACEMENTS FOR THE IBMV2 BENCHMARKS [31]. ALL ROUTED WIRELENGTHS (RWL) ARE IN METERS. "TIME" REPRESENTS ROUTING RUNTIME IN MINUTES. WSA SHOWS IMPROVEMENT ON SOME OF OUR PLACEMENTS, BUT INCREASES ROUTED WIRELENGTH AND VIA COUNTS ON THE LARGEST BENCHMARKS. THE DETAIL PLACERS OF DRAGON 4.0 AND FENGSHUI 5.1 DECREASE THE ROUTABILITY OF OUR PLACEMENTS BY INCREASING RWL AND VIA COUNT ON ALL BENCHMARKS AND THE ADDITION OF VIOLATIONS. BEST LEGAL RWL AND VIA COUNTS ARE IN BOLD.

Benchmark	ROOSTER + NanoRoute				ROOSTER (w/o row orient) + NanoRoute				AmoebaPlace + NanoRoute			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
aes_core	1.339	**125939**	2	32	**1.271**	126645	1	50	1.657	131049	1	28
ethernet	7.287	467777	1	27	**6.145**	**413323**	2	257	7.745	471800	1	28
mem_ctrl	1.061	**87276**	0	22	**0.890**	89153	0	33	1.224	90067	0	21
pci_bridge32	1.336	**114880**	0	35	**1.176**	115675	0	59	1.598	117326	2	35
usb_funct	0.995	**84717**	0	19	**0.860**	85329	0	33	1.106	85739	0	19
vga_lcd	25.906	1131591	2	57	**24.447**	1083504	1	173	25.405	**1076178**	2	90
Ratio	1.000	1.000			0.885	0.979			1.120	1.011		

TABLE X

A COMPARISON OF ROOSTER TO CADENCE AMOEBAPLACE ON THE IWLS 2005 BENCHMARKS [17]. ALL ROUTED WIRELENGTHS (RWL) ARE IN METERS. "TIME" REPRESENTS ROUTING RUNTIME IN MINUTES. ROOSTER IS OUTPERFORMS AMOEBAPLACE BY 12.0% IN RWL AND 1.1% IN VIA COUNTS (WITHOUT ORIENTATION CONSTRAINTS THE IMPROVEMENTS ARE 26.5% AND 3.2%, RESPECTIVELY). BEST RWL AND VIA COUNTS ARE IN BOLD.



(a) DMA (b) DSP1 (c) DSP2 (d) RISC1 (e) RISC2

Fig. 6. The ICCAD'04-Faraday benchmarks (with macros fixed) placed by ROOSTER. Objects with double outlines are fixed.

of the benchmarks can be found in Appendix A of [1]) are shown in Table XI. Results for APlace 2.04 and mPL-R are not shown as they crashed on all but the DMA benchmark (the only Faraday benchmark without macros). Compared to the Silicon Ensemble Ultra placements, ROOSTER improves routed WL by 11.2% and via counts by 4.8%.

Previous work has compared mPL-R/WSA and APlace with Cadence QPlace and found mPL-R/WSA to have the best results on IBMv2 benchmarks [23]. Since we now show better results than mPL-R/WSA, ROOSTER should also compare favorably with QPlace on the IBMv2 benchmarks. Capo has demonstrated comparable performance to QPlace on another set of industry benchmarks [5]. Since ROOSTER considerably improves upon Capo, we expect similar improvements over QPlace as well.

We also performed placement experiments on the IWLS 2005 benchmarks [17]. Unlike the IBMv2 benchmarks which use a 0.25 μm cell library, the IWLS 2005 benchmarks use a Cadence 0.18 μm library. Table X compares ROOSTER with Cadence AmoebaPlace from SOC Encounter 4.1 on a few of the IWLS 2005 designs. All of the benchmarks were routed with Cadence NanoRoute. The two sets of results for ROOSTER differ in how they handle cell orientations in rows that have nontrivial orientations. A full discussion on the orientations of standard cells

Bench-mark	ROOSTER				Silicon Ensemble Ultra v5.4.126			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
DMA	**0.554**	**116414**	0	3	0.644	125328	0	3
DSP1	**1.110**	209274	0	5	1.224	**204863**	0	6
DSP2	**1.067**	**194971**	0	6	1.230	207521	0	6
RISC1	**1.868**	**328699**	5	9	1.957	345615	4	6
RISC2	**1.786**	**324278**	5	7	1.959	347515	2	5
Ratio	1.000	1.000			1.112	1.048		

TABLE XI

ROUTING RESULTS ON THE FARADAY BENCHMARKS WITH MOVABLE MACRO BLOCKS FIXED [1]. ALL ROUTED WIRELENGTHS (RWL) ARE IN METERS. "TIME" REPRESENTS ROUTING RUNTIME IN MINUTES. BEST RWL AND VIA COUNTS ARE HIGHLIGHTED IN BOLD.

Bench-mark	HPWL replaced by MST				StWL replaced by MST			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
ibm01e	0.768	149073	40	188	0.754	136724	2	54
ibm01h	0.768	161339	121	231	0.764	157896	32	184
ibm02e	2.017	281313	2	18	**2.012**	**254610**	0	16
ibm02h	2.010	288491	9	48	2.185	312547	119	89
ibm07e	4.105	481189	0	26	4.102	**475751**	0	26
ibm07h	4.410	528926	18	44	4.214	527378	20	63
ibm08e	4.327	564834	0	28	4.301	**559318**	0	27
ibm08h	4.328	580717	0	33	4.395	618671	4	34
ibm09e	3.192	470294	0	17	3.267	470715	0	18
ibm09h	**3.150**	**475043**	0	18	3.230	478005	0	19
ibm10e	**6.283**	**746000**	0	32	6.538	794192	1	36
ibm10h	6.577	766170	0	38	**6.559**	**765255**	0	37
ibm11e	4.784	608935	0	25	4.798	608887	0	24
ibm11h	4.719	620048	0	24	4.750	619988	0	25
ibm12e	9.277	926201	0	64	9.347	916887	0	55
ibm12h	9.267	991382	1	57	9.301	980202	1	52
Ratio	1.007	1.051			1.015	1.050		

TABLE XII

THE IMPACT OF REPLACING HPWL (FOR HIGH DEGREE NETS) AND STWL (FOR ALL NETS) WITH MST AS THE WIRELENGTH EVALUATOR FOR ROOSTER ON THE IBMV2 BENCHMARKS. ALL ROUTED WIRELENGTHS (RWL) ARE IN METERS. "TIME" REPRESENTS ROUTING RUNTIME IN MINUTES. THE RATIOS ARE WITH RESPECT TO ROOSTER'S PERFORMANCE DESCRIBED IN TABLE VII. LEGAL IMPROVEMENTS TO ROOSTER IN RWL AND VIA COUNTS ARE HIGHLIGHTED IN BOLD.

and pin access is beyond the scope of this work, but the version of ROOSTER that does not respect nontrivial row orientations takes much longer to route than the version that does but can achieve significantly smaller routed wirelengths. ROOSTER improves upon AmoebaPlace in rWL by 12.0% and 1.1% in via count. This empirical comparison to a placement tool from Cadence also suggests that our techniques are superior to those published by Cadence in 1994 [11]. We did not have success using APlace 2.04 and mPl-R on these designs. APlace 2.04 completed global placement on all but the largest benchmark, but terminated with an error message during legalization. mPL-R crashed on all of the benchmarks that were tried.

To see if the routed wirelength of ROOSTER placements could be improved without dramatically increasing its runtime, we attempted to add Minimal Spanning Tree (MST) wirelength into the ROOSTER framework. Recall that if a net has more than a certain threshold of pins, 20 for our experiments, ROOSTER uses HPWL to evaluate the net instead of a Steiner evaluator for

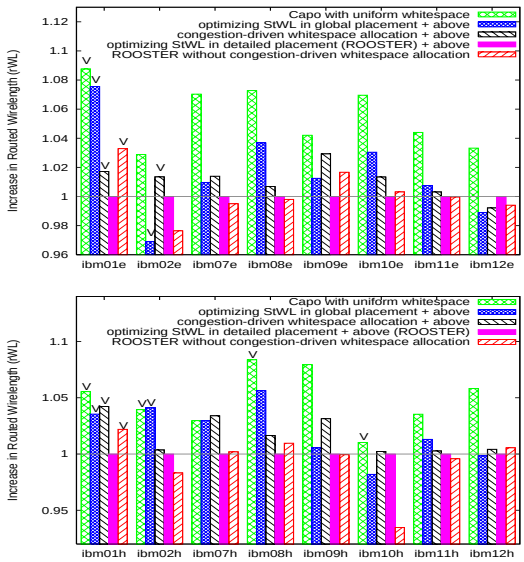


Fig. 7. Impact of individual optimizations on the rWL produced by ROOSTER. "V" indicates violations in routing.

reasons of speed. As MST wirelength is a more accurate estimator of routed wirelength than HPWL and is faster to calculate than StWL, we replaced HPWL with MST wirelength for large nets when calculating weights for partitioning.

Results of adding MST into ROOSTER are shown in Table XII. As we can see, using MST in place of HPWL in ROOSTER increases rWL by 0.7% and via count by 5.1% while reducing routability as 6 benchmarks have violations. Since the fidelity of wirelength evaluator is crucial, we performed an additional experiment where all net weights were calculated using MST WL. Table XII that this increases rWL by 1.5% and via count by 5.0% and reduces routability on 7 benchmarks. These results reinforce our hypothesis that Steiner WL is a better placement optimization objective than MST wirelength.

VIII. CONCLUSIONS AND FURTHER WORK

We have presented techniques which leverage recent advances in RSMT construction [12], [19] to optimize Steiner wirelength in global and detail placement with only a modest increase in runtime, which are currently usable only in our placement algorithm ROOSTER which is freely available as part of the UMpack (http://vlsicad.eecs.umich.edu/BK/PDtools/). As the results of Figure 7 show, the optimization of Steiner tree lengths in global placement is the main source of improved wirelength. However, whitespace distribution is critical to prevent routing violations, even at the cost of increased wirelength. ROOSTER outperforms best published routed wirelength results for Dragon, Capo, FengShui, mPL-R/WSA and APlace 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 others 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, our detail placement window sizes can increase.

Properly accounting for obstacles in placement is an area that 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 [12] and FastSteiner [19]) can handle arbitrary per unit-costs of horizontal and vertical wires. This may provide a safer 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 tests).

Our technique may conceivably be extended to improve circuit timing — this 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.

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

REFERENCES

[1] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa and I. L. Markov, "Unification of Partitioning, Placement and Floorplanning," *ICCAD*, pp. 550-557, 2004.

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

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

[4] A. E. Caldwell, A. B. Kahng, S. Mantik, I. L. Markov and A. Zelikovsky, "On Wirelength Estimations for Row-Based Placement", *IEEE TCAD*, vol. 18, no. 9, pp. 1265-1278, 1999.

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

[6] 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.

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

[8] H. H. Chan, S. N. Adya and I. L. Markov, "Are Floorplan Representations Useful in Digital Design?", *ISPD*, pp. 129-136, San Francisco, 2005.

[9] C. C. Chang, J. Cong, M. Romesis and M. Xie, "Optimality and Scalability Study of Existing Placement Algorithms," *IEEE TCAD*, pp. 537-549, 2004.

[10] 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, 2005.

[11] C.-L. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling," *ICCAD*, pp. 690-695, 1994.

[12] C. C. N. Chu and Y.-C. Wong, "Fast and Accurate Rectilinear Steiner Minimal Tree Algorithm for VLSI Design," *ISPD*, pp. 28-35, 2005.

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

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

[15] 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.

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

[17] IWLS 2005 Benchmarks, `http://iwls.org/iwls2005/benchmarks.html`

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

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

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

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

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

[23] 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.

[24] J. A. Roy, S. N. Adya, D. A. Papa and I. L. Markov, "Min-cut Floorplacement," *IEEE TCAD*, vol. 25, no. 7, pp. 1313-1326, 2006.

[25] J. A. Roy, J. F. Lu and I. L. Markov, "Seeing the Forest and the Trees: Steiner Wirelength Optimization in Placement," *ISPD*, pp. 78-85, San Jose, CA, 2006.

[26] 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.

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

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

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

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

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

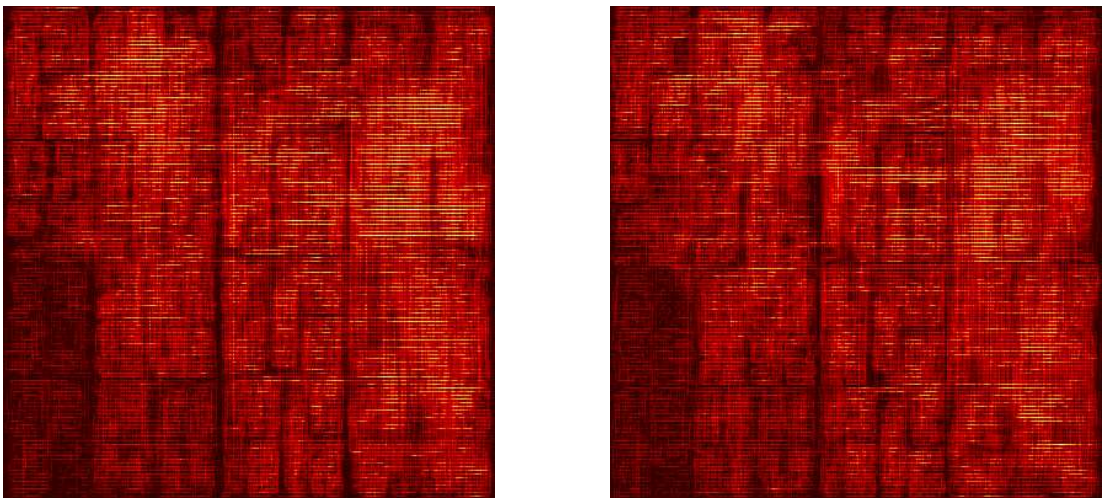


Fig. 8. Congestion maps for the ibm01h benchmark: uniform whitespace allocation (produced with Capo -uniformWS) is illustrated on the left, congestion-driven allocation in ROOSTER is illustrated on the right. The peak congestion when using uniform whitespace is 50% greater than that for our technique. 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.

APPENDIX. STEINER-TREE EVALUATORS: 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) [20], FastSteiner [19] and FLUTE [12]. 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 XIII 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.

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 XIII

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.

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.