Constructive Benchmarking for Placement

David. A. Papa
The University of Michigan
Department of EECS
Ann Arbor, MI, 48109-2122
iamyou@eecs.umich.edu

Saurabh N. Adya
The University of Michigan
Department of EECS
Ann Arbor, MI, 48109-2122
sadya@eecs.umich.edu

Igor L. Markov
The University of Michigan
Department of EECS
Ann Arbor, MI, 48109-2122
imarkov@eecs.umich.edu

ABSTRACT

In the last 20 years, mainstream research in VLSI placement has been driven by formal optimization and the ad hoc requirement that downstream tools, particularly routers, work. Progress is currently measured by improving routed wirelength and place-and-route runtime on large benchmarks. However, these results now appear questionable as (i) major placers were shown to be tuned to particular benchmark suites, and (ii) some reported improvements could not be replicated on full-fledged industrial circuits.

Instead of blind wirelength minimization, our work seeks a better understanding of what a good placer *should* produce and what existing placers *actually* produce. We abstract away details from various circuit patterns into separate "constructive benchmarks" and perform a detailed study of leading placers. Unlike the randomized PEKO benchmarks, ours are highly structured and easy to visualize. We know *all* of their wirelength-optimal solutions, and in many cases there is only one per benchmark. By comparing actual solutions to optimal ones, we reason about the underlying placer algorithms and their possible improvements.

In a new development, we show that the (wirelength) sub-optimality ratio of several existing placers quickly grows with the size of the netlist. Some of the reasons for such poor performance are obvious from our visualizations. While it seems easy to coerce a given placer to improve wirelength on any particular constructive benchmark, improving the overall performance is more difficult. We improve the performance of Capo placer on several constructive benchmarks and a proprietary 72K-cell circuit from IBM, without wirelength penalty on commonly used benchmarks.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing, Layout*

General Terms

Experimentation, Design, Algorithms, Measurement

Keywords

Placer, Benchmark, Performance, Comparison, Evaluation

1. INTRODUCTION

Problems solved by modern placement tools are often computationally intractable from the worst-case perspective, and algorithm developers have to resort to heuristics because finding optimal solutions is not necessary. However, even the best implementations are sometimes over 40% away from optima [11], and commercial tools often place regular datapath-like circuits much worse than human circuit designers [12].

In placement literature, empirical comparisons are typically performed on large irregular circuits, where it is difficult to visually distinguish good placements from poor ones. In order to estimate possible future improvements, several authors propose to evaluate the optimality of existing placers on artificial benchmarks with known solutions or known scaling of optimal wirelength [16, 11]. Sub-optimality gaps have been shown ranging from 40% to over 100% in terms of half-perimeter wirelength [11]. Since artificial benchmarks are not limited by size and can be scaled up astronomically, it is possible to show that currently-achievable placements of larger circuits exhibit growing sub-optimality. However, for some placers sub-optimality increases slowly and is much less than 3x even on benchmarks with millions of movable objects.

Needless to say, good performance on artificial benchmarks does not guarantee good performance on industrial circuits. However, we show that the artificial benchmarks presented in this work can be thought of as benchmarks constructed, by abstracting away some of the details of actual circuit elements found in many circuits today. Improving performance on these constructive benchmarks with abstract features can certainly lead to improvements on fully detailed industrial circuits.

To this end the work in [11] suggests using PEKO benchmarks ("Placement Examples with Known Optima"). The optima for these are not necessarily unique, but they do independently minimize the length of *each* net. These benchmarks are more realistic as they match the net-degree distribution of well-known ISPD 98 net lists released by IBM, but there is no simple expression for exact optimal wirelength of PEKO benchmarks. Furthermore no clusters of logic exist in these benchmarks, and they are unaffected by global placement. Grid netlists [2] and the PEKO benchmarks are publicly available through the GSRC Bookshelf [9], and no existing placer comes close to optimal solutions on PEKO benchmarks. Moreover, it is unclear how to constructively improve the performance of a given placer on those benchmarks. In contrast, due to easy visual analysis of placer behavior on grids, such constructive improvements have been made in Capo [2]. We continue this work by giving a method to construct benchmarks amenable to such analysis.

A serious problem with performance evaluation on large, hard-to-interpret benchmarks is brought up in [1]. The authors empirically show that most of the major academic placers have been tuned to particular benchmarks, often released by the same group. They also observe that many reported improvements cannot be succinctly explained in general terms and do not immediately carry over to real-world benchmarks — sometimes because of feature-limited parsers, sometimes because academic implementations do not support vital layout features, and sometimes for unknown reasons.[1] The authors of [1] advocate (i) placer evaluation on multiple benchmark suites, and (ii) the release of new benchmarks reflecting the sophistication of modern layouts. Indeed, an industry-sponsored group is currently working on such a new benchmark release, but this may not contribute to a better understanding of *how* good placements of large net lists differ from poor placements. Thus, it will be still difficult to describe and improve the performance of existing placers.

The main contribution of our work is a method for constructing artificial scalable benchmarks by abstracting away details from realistic circuit examples. For these benchmarks *all* optimal solutions should be known and be visually comparable to actual placements produced by existing tools. We show that on example benchmarks constructed using this method, all existing placers exhibit sub-optimality ratios that grow linearly with the number of movable objects and associate the most extreme cases with a popular placement strategy. We point out that decision-based placers (e.g., min-cut) can be improved through visual analysis of individual decisions — if a decision does not lead to an optimal placement, the decision-making process should be refined. This process is validated by improving the Capo placer [6] on our benchmarks. Our changes do not significantly affect Capo's performance on popular open benchmarks, but improve wirelength on a 72K-cell fixed-die benchmark with a large amount of whitespace that we received from IBM.

The remainder of this paper is organized as follows. New benchmarks are described in Section 2, and empirical performance of existing placers is reported in Section 3. In Section 4 we improve the Capo placer by comparing its actual placements with optimal ones, conclusions are drawn in Section 5.

2. BENCHMARK CONSTRUCTION

Traditional placement research focuses on high-utilization (low-whitespace) designs, e.g., IBMv1, IBMv2, and PEKO benchmarks have on the order of 15% whitespace. However, many designs today have surprisingly high amounts of whitespace, as illustrated and explained in [4, 2]. In this new context, global positioning of cells is as important as their relative placement, and [4] proposes such techniques by means of analytical constraint generation (ACG) during top-down min-cut partitioning. To capture industrial trends in VLSI design, many of our benchmarks have large amounts of whitespace.

Our benchmarks are constructed by considering tightly connected circuit elements and clustering them together. The nets within these clusters are ignored because we assume a good placement of the individual cells will shorten these nets as much as possible. We do, however, consider the nets which run between the clusters, as they will be affected by the global placement. These nets may be numerous – efficiently modeled by edge weights in a hypergraph. The ultimate goal, however, is to capture key features of the original circuit. We can simply scale heavy weights down, and ignore nets with negligible weight, as well as completely disconnected components. Thus realistic features of actual circuits can be extracted from rest of the low-level details. Figure 1 gives an example of how tightly connected circuit elements can be tied to peripheral I/O pads. By placing and visualizing the result, we can learn a great deal about the quality of a placer and the algorithms it uses.

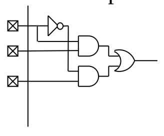


Figure 1:

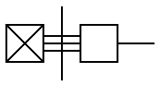


Figure 2:

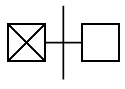


Figure 3:

Each benchmark type seeks to test a particular capability of circuit placers rather than provide a complete design example. A human layout engineer can easily find optimal solutions of all benchmarks, but many academic placers we tried produced unexpectedly poor placements in many cases. We additionally hope to make it visually clear that *if a placer performs poorly on some of our benchmarks*, it may be vulnerable on some realistic designs as well (but not necessarily the other way around). Given that the performance of some existing academic placers is visibly upset by the presence of differently-sized cells, most of our constructive benchmarks use 1x1 cells only. We test for performance with differently sized cells separately. Figure 4 illustrates scalable benchmarks considered in our work. Each benchmark type is described in more detail below, with all of its optimal placements discussed (in many cases the optima are unique).

Peripheral I/O (a): fixed terminals are placed on the periphery of the core area. A movable cell is tied via a two-pin edge to each fixed terminal along the boundaries of the core area. This design emulates a circuit with tightly connected logic tied strongly to the I/O pins. The unique optimal solution entails placing each movable cell adjacent to the terminal it is tied to. This optimal solution is unique because every net achieves minimum possible wirelength, and one end of the net is fixed. The simple strategy of placing every cell, one by one, so as to minimize the total length (linear or squared) of adjacent nets will find the optimal placement in one linear-time pass. However, we are going to show that existing VLSI placers are less successful. In principle, a poor global placement of this benchmark may be vastly improved by a detail placer that moves one cell at a time, but the success depends on the given global placement. This benchmark is parameterized by the height and width of the core area, and optimal wirelength is given by $2*(height+width)$.

Area Array I/O (b): is constructed similarly to (a), with fixed I/O pads arranged in an array throughout the core area. Non-trivial pin offsets force a unique optimal solution in which each net achieves minimum possible wirelength. This benchmark checks placers for compatibility with flip-chip packaging and clusters of logic connected to I/O pins (weaker connections are abstracted away). It is parameterized by the height and the width of the I/O array, the optimal wirelength is given by $1.5*height*width$, due to pin offsets.

Cross (c): is a large cross connecting four groups of fixed peripheral terminals. The cells are arranged and connected as a grid. This design typically has a great deal of whitespace, optimally placed in the corners of the core area. This benchmark represents a datapath like circuit on a timing critical path. Every net in this design achieves minimum possible wirelength, making the placement optimal. Uniqueness can be proven by considering lower bounds for paths or nets that are entirely vertical or entirely horizontal in this optimal placement — any other placement would not match some of these lower bounds. This design is parameterized by the height and width of the core area, the height and width of the arms of the cross, and the horizontal and vertical offset for the arms of the cross. The optimal wirelength is given by $cross\_height*(width+1)+cross\_width*(height+1)+(height-cross\_height)*(cross\_width-1)+(width-cross\_width)*(cross\_height-1)$.

Blob (d): is similar to the cross, but now we only keep the intersection of the two arms of the cross. The movable cells are tied to the boundaries of the core area in the center by long nets. This grid-like structure could be some dense piece of logic such as a multiplier. The cells in the center are placed optimally, because every net in the center has minimum possible wirelength. In both the vertical and horizontal directions paths of nets connect top to bottom,

[1]Note that popular benchmarks from the placement literature are not entirely real-world layout instances. They often make unrealistic simplifying assumptions to make them easier to place.

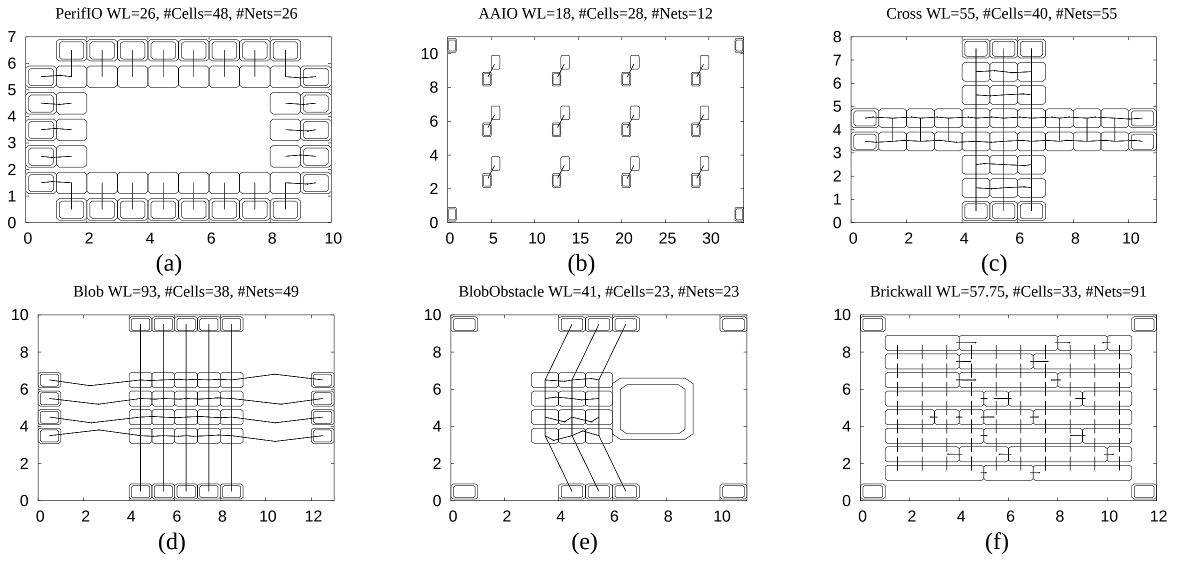


Figure 4: Benchmark types used in our experiments. Fixed cells are shown with double lines. Horizontal connections are intentionally drawn with broken lines. All cells are 1x1 except in (f). The four disconnected corner pads in some benchmarks are used as a visualization aid and do not affect movable cells. A better placement of (e) is possible, if wires are allowed to cross the obstacle.

and left to right. In any other placement, a path is no shorter than the distance between fixed end-terminals of the path. A placement that achieves all these lower bounds must be optimal, observe that there is only one such placement. This design is parameterized by height and width of the core area, height and width of the blob, and vertical and horizontal offset for the blob. The optimal wirelength can be given by $(height+1)*blob\_width+(width+1)*blob\_height$.

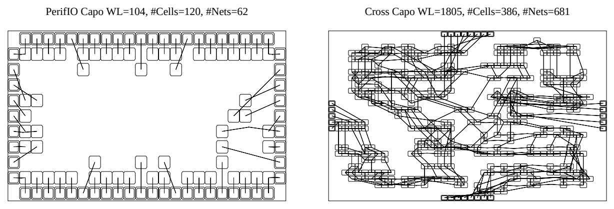
BlobObstacle (e): is a combination of dense logic from (d) and a large immovable obstacle, representing a large macro, where the obstacle forces the blob to be placed off the center. This benchmark challenges the placer to find legal sites without losing much wirelength. Optimal solutions are in a central vertical band, with smallest offsets from the center. Depending on whether the obstacle is interpreted as porous or not (i.e., allowing transit wires to be routed through it), some placements may incur a greater wirelength after routing. If the porous case, the blob may be split vertically and placed above and below the obstacle. The optimal wirelength is then given by $(height+1)*blob\_width$.

Brickwall (f): is constructed from a 0%-whitespace regular grid shown in the Introduction by merging random pairs of neighboring cells and removing connecting wires. This produces cells of varying sizes, but a more controlled construction produces regular brickwalls with identical $1\times n$ bricks. We use irregular brickwalls (see Figure 4) in which two-pin nets connect neighboring bricks horizontally and vertically. These benchmarks test the placement of densely-packed layouts where cells are not $1x1$ (as assumed by early versions of several academic placers). Additionally, irregular brickwalls drastically reduce the possibility of permuting cells to improve wire-lengths — a straightforward optimization technique that does not apply to realistic net lists. Brickwalls have unique solutions up to symmetry (because fixed pads are disconnected). While the optimality of constructed placements is due to each net's achieving optimal length, our randomized construction prevents exact analytical formulas for optimal wirelength.

3. PERFORMANCE OF EXISTING TOOLS

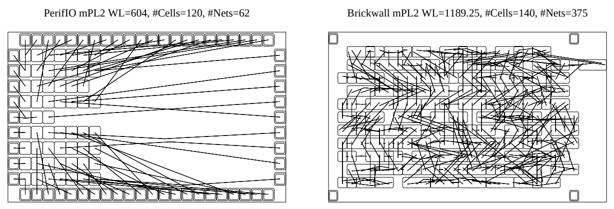
In this section we examine empirical performance of existing academic placement tools, first by placer and then by benchmark. We also study how sub-optimality grows in the number of movable cells.

Capo8.7 [6, 2] On Cross benchmarks spiral patterns are seen (Figure 5(b)), potentially due to how cells are clustered. On AAIO



(a) Capo8.7 placement of **PIO**. Optimal-WL=62.
(b) Capo8.7 placed **Cross**. Optimal-WL=681.

Figure 5: Capo8.7 placements.



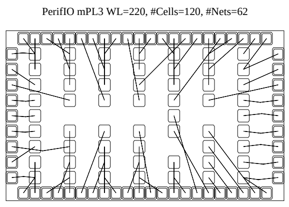
(a) mPL2 packs to the left (**PIO**). Optimal-WL=62.
(b) mPL2 **Brickwall** with overlaps and boundaries exceeded. Optimal-WL=281.75.

Figure 6: mPL2 placements.

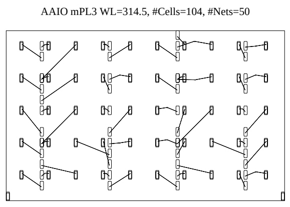
and Blob benchmarks, even distribution of whitespace causes highly sub-optimal results. On the PIO benchmark Capo produces reasonable results (Figure 5(a)), but sub-optimality grows with the size of benchmarks. Capo has trouble finding legal solutions on Brickwall benchmarks due to zero whitespace. Capo and FengShui2.1 are the only placers tested so far that produce legal solutions on the BlobObstacle benchmark. From the AAIO benchmarks, it is clear that improvements can be made to the partitioner used in Capo, because some movables are placed very far from their fixed terminal.

mPL2 [10] This placer packs everything to the left as seen in Figure 6(a) and produces overlapping cells on Brickwall (Figure 6(b)), and BlobObstacle benchmarks. Left-packing causes highly sub-optimal solutions for the Cross, AAIO and PIO suites, – indeed this strategy dominates the performance of mPL2.

mPL3 [8] This newer version of the mPL series has a strong tendency to place cells in columns that span the entire core area — see Figure 7(a)(b). While this is an improvement over the previous

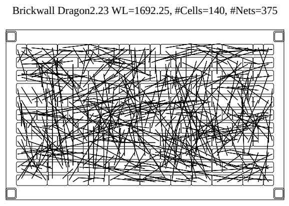


(a) mPL3 places in columns (**PIO**). Optimal-WL=62.

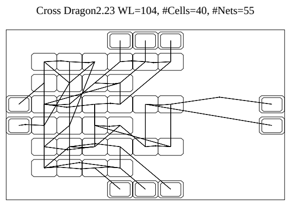


(b) mPL3 **AAIO** with overlaps. Optimal-WL=75.

Figure 7: mPL3 Placements.

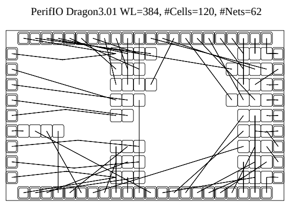


(a) Dragon2.23 on **Brickwall**. Legal placement, not observed from any other placer. Optimal-WL=218.75.

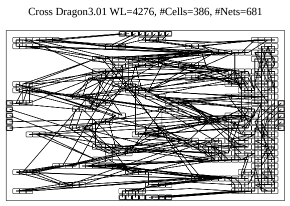


(b) Dragon2.23 on **Cross**. Optimal-WL=55.

Figure 8: Placements by Dragon2.23 in variable-die mode.

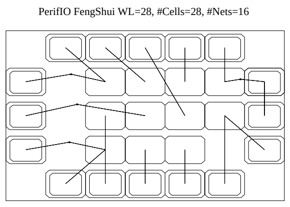


(a) Dragon3.01 in fixed-die mode on **PIO**. Optimal-WL=62.

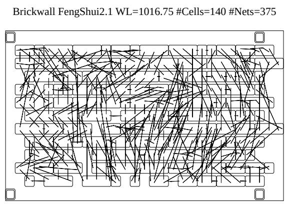


(b) Dragon3.01 on **Cross** placed in fixed-die mode. Optimal-WL=681.

Figure 9: Placements produced by Dragon3.01.

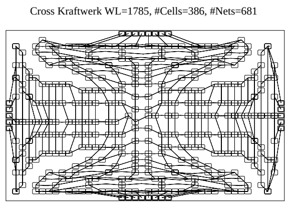


(a) FengShui2.1 **PIO**, this placement is illegal because of overlap. Optimal-WL=16.

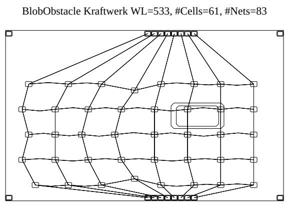


(b) **Brickwall** with no overlap, but cells placed beyond the core area. Optimal-WL=218.75.

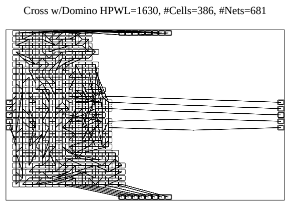
Figure 10: FengShui2.1 Placements.



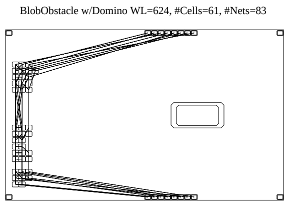
(a) Actual placement resembles Optimal. Optimal-WL=681.



(b) Ignored obstacle and evenly spread cells. Optimal-WL=208.



(c) Domino packs to the left (**Cross**) and produces illegal placement. Optimal-WL=681.



(d) **BlobObstacle** after Domino, legal now. Optimal-WL=208.

Figure 11: Kraftwerk and Kraftwerk+Domino Placements.

strategy of packing everything to the left, it still produces highly sub-optimal solutions in the same benchmarks as before. Cell overlaps are observed on all except Cross benchmarks.

Dragon2.23 [19, 20] We were forced to run this version of the Dragon placer in variable-die mode [19], because it aborts without diagnostics in fixed-die mode [20]. Unfortunately this caused it to pack everything to the left (Figure 8(b)), with wirelength problems as seen previously. However, this placer is the only one to produce legal solutions on the Brickwall benchmark — see Figure 8(a).

Dragon3.01 This newer version of the Dragon series of placers does work in fixed die mode as shown in Figure 9(b). It now takes the strategy of packing most cells to the right, see Figure 9(a)(b). On some benchmarks it leaves islands of cells that are not packed, which improves over Dragon2.23. On at least one Brickwall benchmark, Dragon placed some cells far beyond the boundaries of the core area.

FengShui2.1 [3] This placer tends to pack cell to the left, as seen in Figure 10(a). It places Brickwall without overlaps Figure 10(c), but some cells end up beyond the core area. This placer also seems to avoid placing cells in the first column (Figure 10(a)) for several benchmark types.

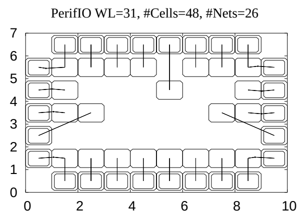
Kraftwerk [14] with Domino [13] Kraftwerk is a global placer that spreads cells around the available area evenly. In order to legalize its placements, one post-processes them with Domino. Illegal Kraftwerk placements are observed in PIO, AAIO, MPIO, Cross, BlobObstacle (Figure 11(b)) and Brickwall benchmarks. In some cases it produces very good results, as in the Cross benchmark in (Figure 11(a)). This placement maintains a cross-like shape akin to its optimal placement. The BlobObstacle benchmark also maintains some semblance of its optimal placement, as seen in Figure 11(b). Domino typically packs all cells to the left (11(c)(d)). We have seen before that this strategy often increases wirelength, but it helps in the case of the Cross, as seen in Figure 11(c).

Asymptotic Growth of Sub-Optimality For all placers tested, the sub-optimality ratio is always greater for larger benchmarks of the same type. Increasing sub-optimality indicates fundamental algorithmic problems, rather than minor implementation oversights. On PEKO benchmarks, sub-optimality grows slowly and is often within 2.5x for benchmarks with a million cells. Yet, in our experiments all placers show a linear growth of sub-optimality in the size of the benchmark, for some benchmark types. The rate varies by placer from 0.006x/cell to 0.200x/cell. Figure 12 shows each placer's sub-optimality and how it grows with benchmark size. Sub-optimality ratios for various benchmarks with ≈ 250 cells are given in Table 1. Earlier pictures suggest that the placers with faster-growing sub-optimality ratios employ a variant of the one-sided packing strategy.

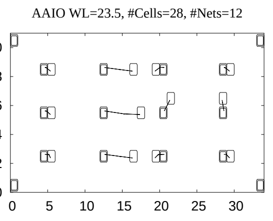
4. IMPROVEMENTS TO CAPO

Results in Section 3 allow us to improve the performance of Capo 8.7 on constructive benchmarks. Since most of these benchmarks are fairly regular, with easily visualizable optimal placements, they are a great aid in debugging a complex placement tool.

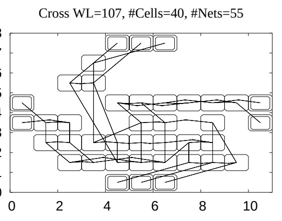
Capo applies a top-down, divide-and-conquer approach to find a global placement. This approach decomposes a given placement in-



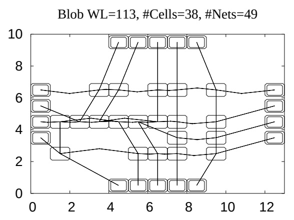
(a) Optimal-WL=26.



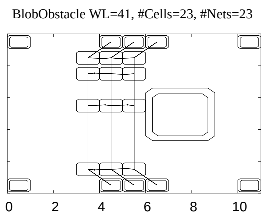
(b) Optimal-WL=18.



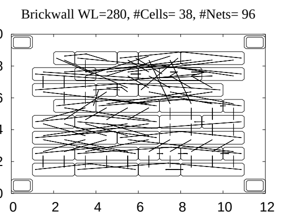
(c) Optimal-WL=55.



(d) Optimal-WL=93.



(e) Optimal-WL=27.



(f) Optimal-WL=59.

Figure 13: Performance of Capo8.8 on all constructive benchmarks. (e) is optimal. (f) is illegal due to overlaps, but much better than Capo8.7 placements. In several cases, all cells are at or near their optimal locations, and a straightforward detail placer would achieve optimal wirelength. Such a detail placer will be added to Capo in the near future.

Benchmarks →	PIO	AAIO	Cross	Blob	Blob-Obstacle	Brick-wall
↓ Placers ↓	↓ Sub-optimality Ratios (times) ↓					
Capo8.7	**6.02**	**6.56**	**2.43**	**1.34**	**1.46**	2.71
Capo8.8	**2.90**	**3.74**	**2.08**	**1.14**	**1.13**	8.93
Dragon2.23	**38.21**	**64.22**	**2.22**	**1.81**	Abort	**6.51**
Dragon3.01	**38.46**	**57.40**	Abort	**2.74**	Abort	6.89
FengShui2.1	**37.37**	**63.64**	**8.34**	**5.80**	**7.60**	3.0e8
Kraftwerk	**16.33**	**2.39**	1.85	**1.36**	1.45	2.45
mPL2	37.87	64.24	**8.34**	5.80	6.49	7.26
mPL3	16.79	4.48	**2.03**	1.47	1.49	6.43

Table 1: Sub-optimality as a ratio of Actual-WL/Optimal-WL. All benchmarks have $250 \pm 3\%$ movable cells. Legal placements are shown in bold-face print. Nearly all illegal placements are due to overlaps. Capo8.8 loses to Capo8.7 by wirelength on Brickwall because it produces nearly-legal solutions.

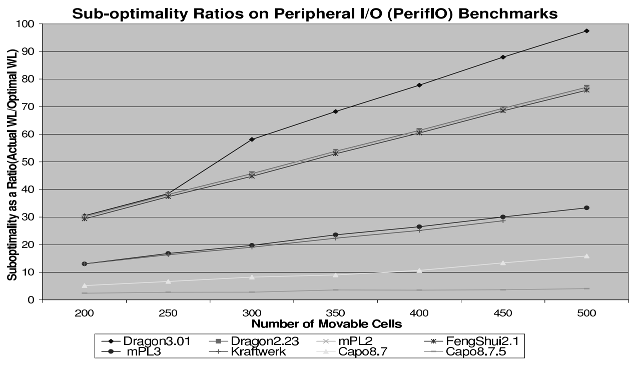


Figure 12: The growth of sub-optimality as a ratio of Actual-WL/Optimal-WL achieved by several placers, as the benchmark size grows – all show a linear growth rate. Placers are mentioned in the order of decreasing sub-optimality ratio. Dragon2.23, FengShui2.1, and mPL2 produce nearly identical results as they all pack cells to the left. A point is missing on the Kraftwerk line because the algorithm did not converge.

stance into smaller instances by subdividing the placement region and assigning cells to sub-regions such that good solutions to sub-instances combine into good solutions of the original instance. The concept of a *placement block* is pivotal. A block represents: 1) a placement region with allowed locations; 2) a collection of cells to be placed in this region; 3) all nets incident to the cells; and 4) locations of all cells beyond the given region that are adjacent to the cells to be placed in the region; such external cells are considered to be terminals, and their locations are fixed. In a min-cut placer like Capo, every placement block yields a hypergraph partitioning instance which is split through min-cut hypergraph bisection with FM-type move-based heuristics. In low-utilization designs often there are only single cells in a placement block during the process of recursive bi-partitioning. Capo8.7 would place this cell at the lower left corner. To improve this behavior, we find the optimal location for the single cell in the placement block which minimizes HPWL.

In the following sub-sections we detail the improvements we made to Capo 8.7 motivated by the constructive benchmarks.

4.1 Whitespace Allocation

First, we explain in brief the whitespace allocation strategy [7] implemented in Capo8.7. Let a placement block have *site area* S, *cell area* C, *absolute whitespace* $W = max S - C, 0$, and *relative whitespace* $w = W/S$. A hypergraph bipartitioning solution implies cell areas C_0 and C_1 in child blocks, such that $C_0 + C_1 = C$, $0 \leq C_0, 0 \leq C_1$. The input to the hypergraph bipartitioner must specify both the netlist and the allowed ranges for C_0 and C_1, i.e., bounds $C_0^{min} \leq C_0 \leq C_0^{max}, C_1^{min} \leq C_1 \leq C_1^{max}$. These bounds establish *absolute* tolerance $T_j = C_j^{max} - C_j^{min}$ and *relative* tolerance $\tau_j = T_j/C$. Capo8.7 uses a mix of fixed tolerances and hierarchical whitespace allocation during top-down placement [7]. The placer chooses vertical or horizontal block splits depending on the blocks' aspect ratio to always cut along the longest side of a block. Vertical partitioning is performed with a fixed 20% tolerance. After partitioning, when the actual total cell area in each partition is available, the vertical cut-line determining the block boundaries is shifted to equalize relative whitespace in the blocks. A different strategy is employed for allocating whitespace for blocks split by horizontal cut-line. During a horizontal split, the partitioning tolerances are calculated based on the relative whitespace of the block and the number of rows in the block. A precise mathematical model of hierarchical whitespace allocation in placement is proposed in [7]. Partitioning tolerances increase as the placer descends to lower levels, and relative whitespace in all blocks is limited from below, thus preventing overlaps.

This facilitates good use of whitespace, when it is scarce and prevents dense regions when large amounts of whitespace are available.

Capo8.7 uses the hierarchical whitespace tolerance calculation only while splitting a block horizontally. The tolerance during the vertical split, is fixed and the vertical cut-line is allowed to move after partitioning to balance the relative whitespace in the two child blocks. This strategy works well for low whitespace designs. However for high whitespace designs, it results in lower tolerances during vertical partitioning and results in excessive wirelength. After studying the behavior of Capo8.7 on the PIO constructive benchmark we added the option -nonUniformWS to Capo8.8. This causes Capo to use the same hierarchical tolerance computation for both horizontal and vertical splits. Since, during the top-down placement process, the aspect ratio of most of the blocks is close to 1.0, we can approximate the number of recursively applied parallel vertical block splits to $n = log_2 R$, where R is the number of rows in the block. With this assumption, the partition tolerances are calculated in the same manner for horizontal and vertical splits. This change allows Capo8.8 to transparently handle designs with a large amount of whitespace. We also test the effect of this change on the "qor" test case from IBM which has 73095 cells ,73155 nets and 74 % whitespace in the design. Capo8.7 distributes the whitespace uniformly around the chip and produces a placement with HPWL of 15.85e6. Capo8.8 allows more tolerance during the initial cuts having the effect of compacting the placement. The final HPWL of the placement produced by Capo8.8 is 10.0e6. The performance of Capo8.8 on the IBMv1 and IBMv2 benchmarks remains unchanged with respect to Capo8.8. This is to be expected as these benchmarks have artificially-created layout regions with a relatively small amount of whitespace.

4.2 Repartitioning with Small Whitespace

While processing the Brickwalls benchmark with 0% whitespace, we realized that Capo8.7 would often produce illegal and overlapping solutions. Accidentally, the built-in naive legalizer of Capo was turned off. Turning the legalizer on after Capo improved the results but Capo was still producing substantial overlaps. In an attempt to reduce the number of overlaps, we revise partitioning in Capo8.8. When a placement block is partitioned, the tolerance is first calculated using the hierarchical tolerance computation described above. If the block is being split vertically, then the vertical cutline is determined to balance the relative whitespace in the two partitions. Furthermore, if the block has very little relative whitespace ($< 2\%$) we then repartition it with a smaller tolerance around the previously determined cut-line. The initial solution to the partitioner is the one obtained from the first partition. This technique was also explored in [2], but caused wirelength degradation on the IBM v1 and v2 benchmarks and was not enabled in Capo8.7 by default.

5. CONCLUSIONS

Our work attempts to complement traditional placement research where improvements are quantified by better routed wirelength on large, hard-to-interpret netlists. While such a metric is certainly necessary for real-world placement hand-off, it is ill-suited for analyzing and improving placement software. We propose a series of artificial, scalable benchmarks each of which captures a particular feature occurring in modern netlists. All optimal placements for each benchmark are known and can be easily visualized. Visual comparisons of actual placements to optimal ones often lead to suggestions for improvement in placement algorithms. To this end, we (i) observe quickly-growing sub-optimalities in solutions produced by Dragon, FengShui, mPL2 and Domino placers, and (ii) trace them to the strategy of packing cells in rows to the left or to the right. Further empirical studies of major academic placers on constructive benchmarks reveal two major approaches to handling whitespace — (i) packing cells in rows, and (ii) distributing whitespace uniformly through the core area. Capo, Kraftwerk and a major industrial placer take the second approach, and tend to produce better wirelengths in our tests. To our knowledge, Capo and Kraftwerk are currently used in industrial EDA tools. We also observe that mPL3 distributes cells in equally-spaced columns, which is certainly closer to uniform distribution than one-sided packing — a departure from mPL2.

Our constructive benchmarks are particularly convenient for debugging placers that perform deterministic and irreversible decisions, e.g., min-cut placers. Indeed, we can track each decision, identify those leading to poor results and refine the decision-making process. In comparison, it may be harder to debug stochastic placers based on reversible iterations with good *expected behavior* because individual moves bear little responsibility for the final results.

In the course of our work, the Capo placer has been considerably improved by the process described above. Wirelengths on constructive benchmarks decreased, with no penalty on existing benchmarks (we observed small improvements in some cases). The benchmarks are available online [15] on the GSRC bookshelf. Ongoing work includes evaluation of industrial placers on constructive benchmarks and several new constructive benchmark families.

6. REFERENCES

[1] S. N. Adya et al., "Benchmarking for Large-scale Placement," *ISPD 2003*, pp. 95-103.

[2] S. N. Adya et al., "On Whitespace and Stability in Mixed-Size Placement and Physical Synthesis," *ICCAD '03*, pp. 311-318

[3] A. Agnihotri, M. C. Yildiz, A. Khatkhate, A. Mathur, S. Ono, P. H. Madden "Fractional Cut: Improved Recursive Bisection Placement," *ICCAD '03*, pp. 307-310

[4] C. J. Alpert, G.-J. Nam and P. G. Villarrubia, "Free Space Management for Cut-Based Placement", *ICCAD* 2002, pp. 746-751.

[5] C. J. Alpert, "The ISPD98 Circuit Benchmark Suite," *ISPD* 1998, pp. 80-85. `http://vlsicad.cs.ucla.edu/~cheese/ispd98.html`

[6] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?" *DAC 2000*, pp.477-82.

[7] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Hierarchical Whitespace Allocation," *IEEE Trans. on CAD*, November 2003.

[8] T. F. Chan, J. Cong, T. Kong, J. R. Shinnerl and K. Sze, "An Enhanced Multilevel Algorithm for Circuit Placement," *ICCAD '03*.

[9] A. E. Caldwell, A. B. Kahng, I. L. Markov, "VLSI CAD Bookshelf" `http://vlsicad.eecs.umich.edu/BK`

[10] C-C. Chang, J. Cong, D. Pan, X. Yuan "Physical Hierarchy Generation with Routing Congestion Control," *ISPD '02*.

[11] J. Cong, M. Romesis, and M. Xie, "Optimality, Scalability and Stability Study of Partitioning and Placement Algorithms",*ISPD* '03, pp. 88-94.

[12] W. J. Dally and A. Chang, "The Role of Custom Design in ASIC Chips", *DAC* 00, pp. 643-647.

[13] K. Doll, F. M. Johannes and K. J. Antreich, "Iterative Placement Improvement By Network Flow Methods". *IEEE Trans. on Computer-Aided Design* , vol.13, (no.10), Oct. 1994. pp. 1189-1200.

[14] H. Eisenmann and F. M. Johannes, "Generic Global Placement and Floorplanning", *DAC* 1998, pp. 269-274.

[15] `http://vlsicad.eecs.umich.edu/BK/FEATURE/`

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

[17] D. J.-H. Huang and A. B. Kahng. "Partitioning Based Standard Cell Global Placement with an Exact Objective," *ISPD* 1997, pp. 18-25.

[18] H. Nakashima, J. Inoue, K. Okada and K. Masu "ULSI Interconnect Length Distribution Model Considering Core Utilization," *DATE* '04

[19] M. Wang, X. Yang and M. Sarrafzadeh, "Dragon2000: Standard-cell Placement Tool for Large Industry Circuits," *ICCAD 2000*, pp. 260-263.

[20] X. Yang, B.-K. Choi and M. Sarrafzadeh, "Routability Driven White Space Allocation for Fixed-Die Standard-Cell Placement," *ISPD 2002*, pp. 42-50.