Solving Hard Instances of Floorplacement

Aaron Ng, Igor L. Markov
Department of EECS
The University of Michigan
1301 Beal Ave
Ann Arbor, MI 48109
{aaronnn, imarkov}@umich.edu

Rajat Aggarwal
Xilinx, Inc.
2100 Logic Dr
San Jose, CA 95124
rajat.aggarwal@xilinx.com

Venky Ramachandran
Calypto Design Systems, Inc.
2933 Bunker Hill Lane
Santa Clara, CA 25054
venkyr@calypto.com

ABSTRACT

Physical Design of modern systems on chip is extremely challenging. Such digital integrated circuits often contain tens of millions of logic gates, intellectual property blocks, embedded memories and custom RTL blocks. At current and future technology nodes, their power and performance are impacted, more than ever, by the placement of their modules. However, our experiments show that traditional techniques for placement and floorplanning, and existing academic tools cannot reliably solve the placement task.

To study this problem, we identify particularly difficult industrial instances and reproduce the failures of existing tools by modifying public-domain netlists. Furthermore, we propose algorithms that facilitate floorplacement of these difficult instances. Empirically, our techniques consistently produced legal placements, and on instances where comparison is possible, reduced wirelength by 3.5% over Capo 9.4 and 14.5% over PATOMA 1.0 — the pre-existing tools that most frequently produced legal placements in our experiments.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids — *placement and routing*; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms: Algorithms, experimentation

Keywords: Circuit layout, placement, floorplanning, floorplacement, benchmarks, RTL

1. INTRODUCTION

As demonstrated by the ISPD 2005 placement contest, automated layout of modern systems-on-chip (SoCs) is very different from traditional sea-of-gates layout, both in scale and sophistication. Such integrated circuits (ICs) may contain tens of millions of logic gates, hundreds of complex configurable intellectual property (IP) blocks, embedded memories that come in different sizes and aspect ratios, as well as custom RTL blocks. Placement plays a key role not only in the final implementation stage, but also in the early stages of the design flow by enabling more accurate interconnect and performance analysis. To be useful in estimation, placement techniques must support very high capacity, complete quickly and be robust across a variety of designs.

To improve capacity and runtime, we observe that at the register-transfer level and above, the design intent is expressed in terms of word-level arithmetic and logical operators within separate parallel hardware threads or "always" blocks in Verilog. It is neither necessary nor even desirable to decompose these operators into bit-level (gate-level) netlists in order to estimate performance, which requires knowing locations. Such a transformation often increases the number of modules by 20-50x, especially for designs with wide datapaths, and complicates interconnect analysis. However, since word-level operators are known in advance, one can pre-characterize their area, timing and power, essentially building an RTL library. During placement, such library components can be viewed as soft blocks of a particular size and shape, subject to certain aspect ratio constraints (with hard blocks being a special case).

The task of finding non-overlapping locations of modules of varying sizes while optimizing for certain objective(s) is usually called floorplanning. However, traditional floorplanning does not scale beyond a few hundred placeable modules. On the other hand, standard-cell placement can deal with millions of instances, except that those instances are assumed to have identical heights and similar lengths, which helps fitting them in equally-spaced rows and sites. These constraints prevent difficult block-packing, while facilitating fast and high-quality detailed placement algorithms. However, their use for RTL entities remains unexplored.

A recent unification of placement and floorplanning termed *floorplacement* [24] seeks to bridge the gap between floorplanning and placement by combining their best features. Floorplacement appears promising for SoC layout because of its high capacity and the ability to pack blocks. However, as our experiments demonstrate, existing tools for floorplacement are fragile — on many instances we tried they fail, or produce remarkably poor placements. This observation sets the stage for our work, in which we make the following contributions:

- We identify hard floorplacement instances derived from industry designs and modify public-domain netlists to behave similarly. We explain why existing algorithms for mixed-size placement and floorplanning fail on hard instances.
- We propose four synergistic techniques for floorplacement that in particular succeed on hard instances: (i) selective floorplanning with macro clustering, (ii) improved obstacle evasion for B*-trees, (iii) ad hoc look-ahead in top-down floorplacement, and (iv) whitespace allocation by density. Obstacle evasion is especially important for top-down floorplacement, even for designs that initially have no obstacles.
- We demonstrate empirical improvements compared to Capo 9.4 and PATOMA 1.0. Our techniques lead to 68% and 36% increase in success rates in floorplacement, and shorter wirelength by 3.5% and 14.5%, respectively.

The rest of this paper is organized as follows. We review current state-of-the-art in mixed-size placement and floorplanning in Section 2, and describe the nature of difficult instances in Section 3. In Section 4 we evaluate existing software tools, show their limitations and identify the weaknesses of core algorithms. Our algorithmic improvements are presented in Section 5 and empirically evaluated in Section 6. Conclusions are summarized in Section 7.

2. PREVIOUS WORK

In this section, we outline the state-of-the-art in mixed-size placement and floorplanning. First, we discuss floorplanning and introduce basic computational techniques for block packing. Then we show a natural progression to hierarchical floorplanning, mixed-size placement and floorplacement.

Common approaches to floorplanning. Modern VLSI floorplanning is predominantly used with fixed outlines [4, 16]. The fixed-outline floorplanning problem seeks non-overlapping locations of modules (blocks) within a fixed outline, subject to optimizing for certain objective(s), such as wirelength, power or performance. The most popular algorithmic framework in floorplanning is Simulated Annealing and is implemented using a certain *floorplan representation* that captures relative locations of modules and is easy to perturb, but can also generate non-overlapping module locations. Simulated annealing is attractive because it allows to optimize a wide variety of objective functions and makes customization very easy. It can deal with hard blocks with any aspect ratios and also soft blocks. It also allows to handle fixed modules, as discussed in Section 5. Sequence-Pair [22] and B*-tree [8] are two popular floorplan representations.

The Sequence-Pair representation consists of two ordered lists, which capture geometric relations between pairs of blocks by the relative ordering of blocks. However, in our work we use the more complicated B*-tree representation which captures a compacted packing by a binary tree, in which each node corresponds to a block. The root of the tree is in the bottom-left corner of the outline, and the packing is compacted in that direction. The root node represents the bottom-left block. A left child is the lowest right neighbor of its parent and a right child is the lowest block above its parent that shares the same x-coordinate as its parent [7]. Given a B*-tree, block locations can be found by a depth-first traversal of the B*-tree. After a block A is placed at (x_A, y_A), its left child L is considered and set $x_L = x_A + w_A$, where w_A is the width of A. Then y_L is the smallest non-negative value such that L avoids overlaps with previously placed blocks. After returning from recursion at L, the right child of A, R is placed at $x_R = x_L$, and y_R is the smallest value possible such that R does not overlap with placed blocks. The *contour* data structure allows one to evaluate B*-tree packings in $O(n)$ time. The two floorplan representations encode large solution spaces: $O(n!2^{2n-2}/n^{1.5})$ for B*-tree and $n!^2$ for Sequence-Pair. According to [7], B*-tree packs somewhat better than Sequence-Pair. Additionally, we found that it better supports the handling of obstacles.

Parquet [1] is a floorplanning framework based on simulated annealing and uses both Sequence-Pair and B*-tree representations. Parquet produces high-quality solutions, but due to the runtime of the simulated annealing framework, it cannot be practically applied on large designs without clustering [24].

Hierarchical floorplanning, mixed-size placement and floorplacement. PATOMA 1.0 [13] pioneered a top-down floorplanning framework that utilizes fast block-packing algorithms (ROB or ZDS [12]) and hypergraph partitioning with hMetis [20]. This approach is fast and scalable, and provides good solutions for many input configurations. Fast block-packing is used in PATOMA to guarantee that a legal packing solution exists, at which point the burden of wirelength minimization is shifted to the hypergraph partitioner. This idea is applied recursively to each of the newly-created partitions. In end-cases, when partitioning cannot be used because it creates unsatisfiable instances of block-packing, block locations are determined by fast block-packing heuristics. The placer PolarBear [14] integrates algorithms from PATOMA to increase the robustness of a top-down min-cut placement flow.

S/W tools	Algorithms	Support for blocks		Availability
		Hard	Soft	
APlace 2.0 [19]	analytic	+	-	binary only
Capo 9.4 [23]	min-cut & annealing	+	+	open source
Dragon [29, 26]	min-cut & annealing	+	-	unavailable*
FastPlace [27]	analytic	+	-	unavailable*
FDP [28]	analytic	+	-	unavailable*
FengShui 5.1 [21]	min-cut & Tetris packing [15]	+	-	binary only
IMF [11]	min-cut & refinement	+	-	binary only
mPL6 [9, 6]	analytic	+	-	unavailable*
PATOMA 1.0 [13]	min-cut & fast packing	+	+	binary only
PolarBear [14]	min-cut & fast packing	+	-	binary only
UPlace [30]	analytic	+	-	unavailable*

* Publicly available binaries for FastPlace, mPL and Dragon do not support hard blocks. Available source code for FDP is outdated relative to published results.

Table 1: The availability of mixed-size placers & floorplanners.

Similar to PATOMA, the floorplanner IMF [11] utilizes top-down partitioning, but allows overlaps in the initial top-down partitioning phase. A bottom-up merging and refinement phase fixes overlaps and further optimizes the solution quality.

The min-cut placer FengShui is based on the *fractional cut* technique [21] that finds tentative locations of macros and standard cells by minimizing wirelength, but allows modules to overlap. Legalization is performed as post-processing, assuming that the amount of overlap is small.

The analytic placer APlace [17] uses an iterative algorithm that optimizes a non-linear objective function. The objective balances wirelength against a density/spreading function that captures overlapping modules. APlace was extended in [18] to handle mixed-size placement by specializing the density function to large macros. Global placements are post-processed by legalization, assuming that the amount of overlap is small.

The Capo software uses hypergraph partitioning in a similar way to PATOMA, but employs a floorplacement flow [24] and relies on an annealing-based floorplanner Parquet [1]. When Parquet cannot solve block-packing, Capo backtracks. Parquet is much slower than ROB and ZDS because it explores a larger fraction of the solution space, but it can optimize wirelength better in larger floorplanning instances. Unlike PATOMA, Capo is based on a standard-cell placer, and can therefore handle both macros and standard cells. Also, in Capo most floorplanning calls determine final locations of macros, whereas in PATOMA at least 50% of floorplanning calls perform look-ahead to guarantee that legal solutions exist.

Several other tools (Dragon, FastPlace, UPlace, FDP and mPL) claim to support mixed-size placement, but their public-domain implementations lag behind published results, do not handle hard blocks, or are entirely unavailable. We therefore cannot use these tools in our work. We also do not evaluate IMF and PolarBear, because IMF does not handle standard cells, which are present in the industrial designs we consider, while PolarBear does not handle soft blocks (but is otherwise similar to PATOMA). Table 1 summarizes published academic tools for mixed-size placement and large-scale interconnect-driven floorplanning.

3. DIFFICULT INSTANCES

Since block packing is NP-hard, a key challenge for heuristics is to moderate their effort (runtime) while ensuring good solution quality on a variety of inputs. To this end, we identified two sets of netlists that appear particularly difficult for all tools we evaluated.

Proprietary instances. The designs, provided by Calypto Design Systems, Inc., include customer chips (e.g., CPUs and video ICs) and internally-generated regression tests [32]. As shown in Table 2, these designs range in size from 81 to 8827 RTL modules, have 20% whitespace, and have no fixed modules except for peripheral I/O pads. Aside from the standard cells present in most designs, all blocks are soft. The main objective of our experiments is to perform RTL placement using as many existing alternative tools as possible. To this end we have access to APlace 2.0, Capo 9.4, FengShui 5.1 and PATOMA 1.0. Of these four tools, APlace 2.0 and FengShui 5.1 do not support soft blocks, so the solution space is simplified for these placers by changing all soft blocks to hard blocks with an aspect ratio of 1.0 (the easiest to pack). However, even when Capo and PATOMA are run on hard-block variants of these designs, their runtimes and Half-Perimeter Wire-Length (HPWL) are only slightly worse, while the results are still better than those of APlace and FengShui, with all other trends reproduced. The proprietary designs consist mostly of macros, and the macros may vary greatly in size. As we show next, these designs upset existing academic tools for mixed-size placement and floorplanning.

Proprietary designs	Movable modules		Nets	$Area_{largest}$ (%)	$Area_{largest}/Area_{smallest}$
	Cells	Macros			
cal040	1	4605	4607	0.1	650
cal098	3200	1212	4673	0.1	529
cal336	17	105	147	2.2	11556
cal353	217	459	908	7.0	11556
cal523	934	1936	4350	0.3	3080
cal542	7	74	92	20.1	11556
cal566	93	1553	5502	1.2	11556
cal583	773	1530	3390	0.4	2916
cal588	293	495	1111	0.6	900
cal643	139	316	598	6.5	6162
calDCT	0	8827	11463	50.0	185330

Table 2: Characteristics of the proprietary designs.

Benchmarks	Movable modules		Nets	$Area_{largest}$ (%)	$Area_{largest}/Area_{smallest}$
	Cells	Macros			
ibm-HB$^+$01	0	911	5829	6.4	8416
ibm-HB$^+$02	0	1471	8508	11.3	3004.3
ibm-HB$^+$03	0	1289	10279	10.8	33088
ibm-HB$^+$04	0	1584	12456	9.2	13296.5
ibm-HB$^+$06	0	749	9963	13.6	18173.8
ibm-HB$^+$07	0	1120	15047	4.8	399.5
ibm-HB$^+$08	0	1269	16075	12.1	50880
ibm-HB$^+$09	0	1113	18913	5.4	29707
ibm-HB$^+$10	0	1595	27508	4.8	71299
ibm-HB$^+$11	0	1497	27477	4.5	9902.3
ibm-HB$^+$12	0	1233	26320	6.4	74256
ibm-HB$^+$13	0	954	27011	4.2	33088
ibm-HB$^+$14	0	1635	43062	2.0	17860
ibm-HB$^+$15	0	1412	52779	11.0	62781.3
ibm-HB$^+$16	0	1091	47821	1.9	31093
ibm-HB$^+$17	0	1442	56517	0.9	12441
ibm-HB$^+$18	0	943	42200	1.0	3384

Table 3: Characteristics of the IBM-HB$^+$ benchmarks.

Our experiments were conducted on a 2.4 GHz Athlon workstation with 3GB RAM. As Capo uses a randomized algorithm, its results are averaged over 3 independent runs. Empirical results in Table 4 demonstrate that many existing tools experience difficulties even with smaller designs, indicating that scalability is not the only problem here. Capo places all designs (some with overlaps), but times out for the largest design *calDCT* with 8827 macros, suggesting that scalability is a serious issue nevertheless. The design *cal040* appears challenging for PATOMA and APlace even though it has a small range of macro sizes, compared to other designs.

Public-domain instances. We reproduce the difficulties observed on hard floorplacement instances by modifying seventeen of the IBM-HB benchmarks released in [13], for further evaluation of difficult mixed-size placement[1]. The IBM-HB [13, 31] benchmarks were generated from the IBM/ISPD'98 suite [3] and contain both hard and soft blocks in a fixed die with 20% whitespace. The soft blocks are clusters of standard cells while the hard blocks represent original macros from the IBM/ISPD98 benchmarks. The benchmarks range from 500 to 2000 blocks in size. Our modification is as follows: the largest hard macro is inflated by 100%, while the areas of the remaining soft macros are reduced to preserve the total cell area. We call this new benchmark family IBM-HB$^+$ [32]. A more detailed future study can involve varying the dimensions of more than one macro at a time, and even more difficult floorplacement problems can be constructed. For now, inflating only the largest macro and shrinking the rest is sufficient to reveal the limitations of existing tools and provides enough food for thought.

4. IDENTIFYING WEAKNESSES IN FLOORPLACEMENT ALGORITHMS

By studying the data and plots collected, as well as logs produced by various tools, we identify shortcomings of published algorithms.

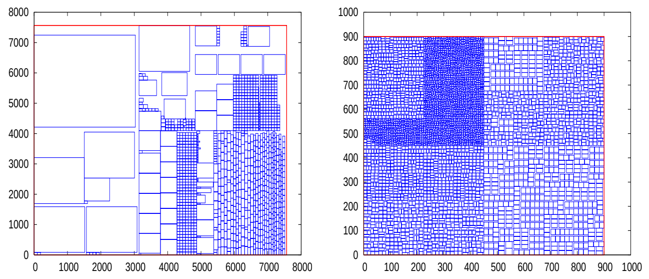

Figure 4: The *IBM-HB$^+$10* benchmark and the *cal040* design are pathological examples for PATOMA 1.0, which produces placements with HPWL that are 1.4x and 9.7x larger (resp.), than the best seen in legal solutions. These examples may be highlighting shortcomings of fast block-packing algorithms, even when the variation of block sizes is relatively small.

PATOMA utilizes partitioning-based algorithms [13] that allow it to solve floorplacement instances very quickly. When a set of floorplanning instances falls within the operating range of PATOMA, resulting solutions are slightly better than Capo's and are found several times faster. However, PATOMA seems to trade robustness for runtime and in some cases generates solutions with remarkably long wires — for example, Figure 4 shows PATOMA solutions for the *IBM-HB$^+$10* benchmark and the *cal040* design, which have 1.4x and 9.7x worse HPWL, respectively, compared to the best seen solutions. This suggests that PATOMA's algorithms do not adequately search the solution space, making PATOMA's performance unpredictable and inconsistent across the benchmarks.

Upon further analysis, the severe degradation in solution quality can be attributed to PATOMA's "guarantors" — fast block-packing heuristics that guarantee *legal* solutions, but do not guarantee *good*

[1]We exclude *IBM-HB05* because it does not contain hard macros.

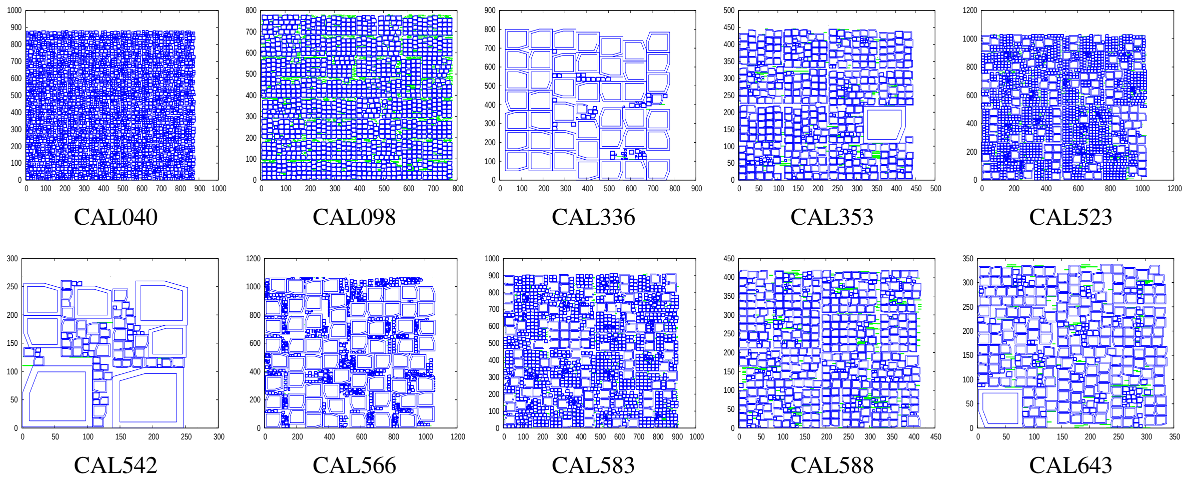

Figure 1: Proprietary designs used in our work. Macros are shown in blue and standard cells in green.

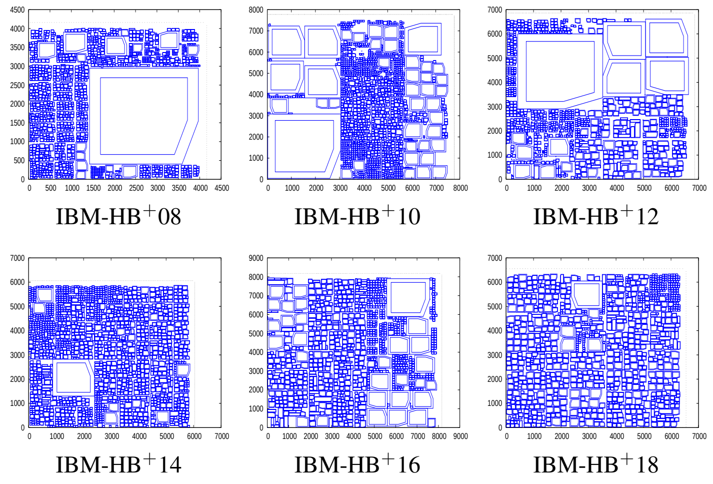

Figure 2: Six of the seventeen IBM-HB$^+$ benchmarks.

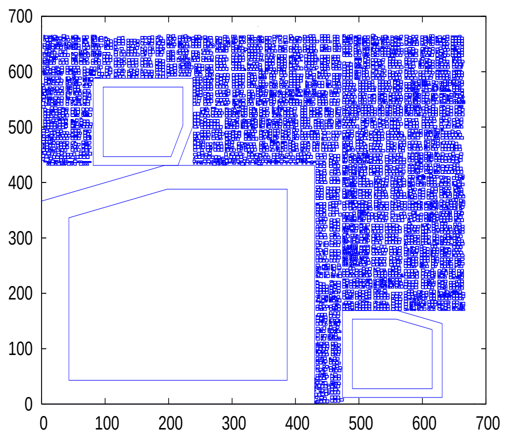

Figure 3: The 8827-macro design *calDCT*.

solutions. Indeed, when PATOMA fails to find a legal partitioning solution, it resorts to the "guaranteed" floorplanning solution produced during look-ahead without sufficient regard to wirelength minimization. PATOMA's results suggest that hard instances expose the gap between PATOMA's predictions and its ability to implement them. Another possibility is that PATOMA's fast guarantors jump the gun and report failures when good packings exist (but take time to find). If either of these effects happens early in the PATOMA flow, PATOMA will produce an essentially random legal placement, as it does on the *cal040* design.

In contrast to PATOMA, Capo uses a much slower simulated-annealing floorplanner (Parquet) in bins (partitions) created during top-down min-cut placement. Parquet can handle up to 50-100 blocks well [7], but then becomes inefficient for larger instances. A built-in clusterer, also requested by Capo, extends Parquet's operating range but impacts solution quality. Capo's floorplacement flow decides to floorplan a bin when a block is too large to fit in either child-bin [24]. Given that large blocks are common in modern designs (e.g., an L2 cache can take 50% of a microprocessor's area), Capo may decide to perform annealing near the top level, on almost all the blocks in the design. When applied to thousands of blocks, Parquet spends an inordinate amount of time and typically

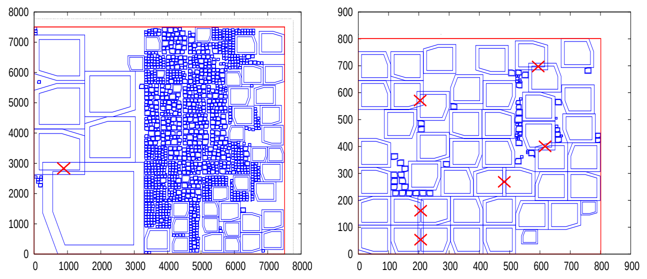

Figure 5: The *IBM-HB$^+$10* benchmark and *cal336* design placed by Capo 9.4. These plots of illegal placements show that partitioning may produce bins that are difficult (or impossible) to floorplan, since the partitioner may mis-approximate the area required by a packing of the blocks. Overlaps between modules are marked with red crosses.

does not find a reasonable solution. According to logs, Capo 9.4 goes through this scenario on the proprietary design *calDCT* that includes a very large block, as shown in Figure 3.

cal bench	PATOMA 1.0			Capo 9.4 -faster			APlace 2.0			FengShui 5.1			SCAMPI (our work)				
	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	HPWL (e+04)	ovlp (%)	time (s)	*vs.* PATOMA (HPWL)	*vs.* CAPO (HPWL)
040	177.2	0.0	9.6	**18.7**	0.0	45.4	20.7	0.3 ⊗	239.0	20.6	0.0	37.9	18.8	0.0	44.9	0.11x	1.00x
098	52.3	0.0	11.2	31.8	1.3	788.2	22.6	0.3	271.6	24.0	0.0 ⊗	6.0	**30.7**	0.0	302.4	0.59x	-
336	**2.8**	0.0	1.2	3.5	9.1	22.5	2.2	0.1 ⊗	83.5	7.6	0.0	0.2	3.3	0.0	30.4	1.20x	-
353	7.6	0.0	1.0	6.5	0.5	52.6	4.6	0.3	211.8	31.5	1.6 ⊗	0.8	**6.3**	0.0	44.5	0.83x	-
523	123.7	0.0	3.4	34.7	0.3	240.2	27.5	0.3	920.3	348.7	0.0	2.8	**37.1**	0.0	460.1	0.30x	-
542	0.9	0.0	0.1	**0.8**	0.0	3.3	0.7	0.1	42.8	×	×	×	**0.8**	0.0	2.4	0.89x	1.00x
566	83.6	0.0	4.9	63.8	1.9	225.7	46.9	0.5	341.1	493.6	3.8 ⊗	3.2	**69.3**	0.0	162.8	0.83x	-
583	47.0	0.0	2.3	26.1	0.6	190.6	20.6	0.2	421.2	×	×	×	**25.1**	0.0	342.6	0.53x	-
588	8.8	0.0	0.7	6.3	1.1	60.4	4.8	0.5	41.5	×	×	×	**6.9**	0.0	102.7	0.78x	-
643	4.9	0.0	0.6	3.8	0.9	18.8	3.0	0.4	29.3	15.3	0.2 ⊗	0.5	**3.7**	0.0	40.0	0.76x	-
DCT	×	×	×	×	×	>1800	33.1	1.7 ⊗	719.4	184.7	0.0	8.0	**37.2**	0.0	123.5	-	-
													Average			0.68x	1.00x

× indicates time-out, crash, or a run completed without producing a solution; ⊗ indicates an out-of-core solution

Table 4: Runs on proprietary designs. Best legal solutions are emphasized in bold.

ibm *-HB*$^+$ bench	PATOMA 1.0			Capo 9.4 -faster			APlace 2.0			FengShui 5.1			SCAMPI (our work)				
	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	HPWL (e+06)	ovlp (%)	time (s)	*vs.* PATOMA (HPWL)	*vs.* CAPO (HPWL)
01	3.9	0.0	5.6	5.4	1.4	651.5	2.7	2.7	68.0	3.0	0.2 ⊗	16.6	**3.4**	0.0	62.0	0.87x	-
02	×	×	×	19.1	0.0	1539.7	5.0	2.6	101.5	8.7	0.9 ⊗	43.6	**8.0**	0.0	139.6	-	0.42x
03	×	×	×	×	×	>1800	7.4	2.1	101.3	×	×	×	**9.5**	0.0	104.6	-	-
04	×	×	×	×	×	>1800	8.2	2.8	113.9	10.8	0.2 ⊗	41.4	**12.3**	0.0	144.1	-	-
06	×	×	×	×	×	>1800	8.2	1.0	122.5	10.7	1.4 ⊗	36.0	**11.0**	0.0	170.0	-	-
07	16.8	0.0	13.6	15.8	0.0	115.31	13.7	1.4	218.4	37.1	0.0	5.1	**15.7**	0.0	99.9	0.93x	0.99x
08	×	×	×	×	×	>1800	16.6	1.0 ⊗	294.2	21.8	0.5 ⊗	60.6	**20.5**	0.0	188.4	-	-
09	×	×	×	20.2	0.2	188.9	15.1	0.9	222.4	20.6	1.2 ⊗	42.9	**22.2**	0.0	182.0	-	-
10	×	×	×	45.9	2.7	263.7	36.9	0.3	529.5	×	×	×	**55.2**	0.0	319.9	-	-
11	**25.3**	0.0	49.2	28.1	0.0	140.5	24.5	1.1	270.3	30.4	0.2 ⊗	63.8	27.8	0.0	144.7	1.10x	0.99x
12	×	×	×	**63.4**	0.0	482.2	×	×	>1800	52.3	0.0 ⊗	39.2	67.6	0.0	406.1	-	1.07x
13	**37.5**	0.0	34.7	39.6	0.0	221.5	31.7	0.5	240.4	×	×	×	42.2	0.0	209.6	1.13x	1.07x
14	68.7	0.0	70.9	68.2	0.0	320.7	57.1	1.0 ⊗	392.9	74.0	2.7	89.7	**66.4**	0.0	268.3	0.97x	0.97x
15	×	×	×	×	×	>1800	87.5	1.5	422.2	90.6	0.0 ⊗	100.3	**88.2**	0.0	375.9	-	-
16	**100.3**	0.0	74.4	106.9	0.0	431.5	89.8	0.3	528.1	×	×	×	106.2	0.0	306.5	1.06x	0.99x
17	**141.4**	0.0	95.9	152.6	0.1	397.1	133.9	0.5	799.3	×	×	×	152.7	0.0	385.7	1.08x	-
18	**72.6**	0.0	67.2	75.9	0.7	220.1	69.1	0.6	344.0	×	×	×	77.8	0.0	192.3	1.07x	-
													Average			1.03x	0.93x

× indicates time-out, crash, or a run completed without producing a solution; ⊗ indicates an out-of-core solution

Table 5: Runs on IBM-HB$^+$. Best legal solutions are emphasized in bold.

Another shortcoming of partitioning-based tools like Capo and PATOMA is that their area-balance calculations rely on the sums of block and cell areas. These do not account for dead-space that is sometimes inevitable around large blocks and is particularly problematic when most blocks appear in one partition and most cells in the other. This scenario plays out in Figure 5:left which shows a placement produced by Capo for *IBM-HB*$^+$*10*. The first cut was vertical with $x_{cutline} = 3300$. The second cut in the resulting partition on the left was horizontal with $y_{cutline} = 4100$. The partition in the lower left corner failed to floorplan, and it was merged with its sibling (i.e., the bottom left partition was merged with the top left partition). However, the merged partition failed to floorplan as well, at which point Capo accepted a bloated floorplan that it could not legalize during post-processing. In other words, Capo 9.4 can only backtrack once. Additionally, we see that a partitioner underestimated the amount of area required by the blocks in the left partition and probably over-estimated the area required by the right partition. This gap between partitioning and floorplanning must be addressed to improve floorplacement on difficult instances.

FengShui apparently assumes that the amount of overlap generated by fractional cut is minimal, and relies on a simple legalizer to produce non-overlapping solutions. This may work for fine-grain mixed-size placement instances, but fails for complex floorplans, as shown in Figure 6. In some cases FengShui 5.1 places blocks out of core and in others produces remarkably high wirelength.

APlace 2.0 also fails to legalize its global-placement solutions – some modules are placed beyond the fixed outline and some overlap, e.g., Figure 7. We believe that all analytic placers are likely to experience similar difficulties unless they use a strong legalizer that can accurately manipulate the shape of every module. Another possible limitation is due to the hierarchical clustering algorithm in APlace 2.0 [19], which prefers to cluster modules and sub-clusters of similar sizes. Having cells of similar sizes in a cluster may be useful for area-estimation during cell-spreading and legalization, but also artificially restricts module locations and could lead to routing congestion.

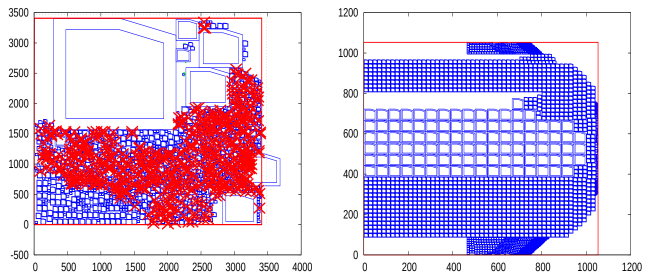

Figure 6: The *IBM-HB*$^+$*04* benchmark and *cal523* design placed by FengShui 5.1. The left placement is illegal because of overlapping modules (marked with red crosses) and a module placed outside the core boundary (boundary shown in red). The right placement is legal but has 5x worse wirelength than what is possible.

In all placers, any problems left after the global placement phase must be repaired during legalization or detail placement. The more overlaps in global placement, the harder it will be to produce legal solutions with low wirelength. Based on our results, it is not clear if the traditional separation into global and detail placement is

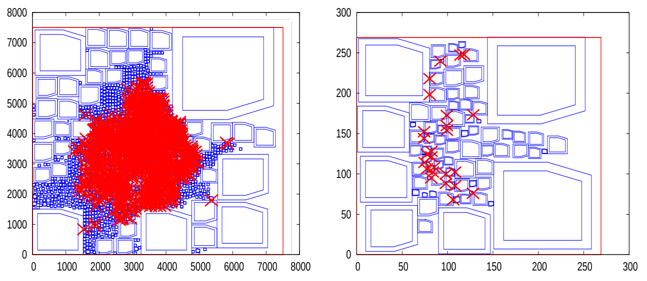

Figure 7: The *IBM-HB$^+$10* benchmark and *cal542* design placed by APlace 2.0. Overlaps between modules are marked with red crosses. These plots of illegal placements indicate that floorplanning and post-placement legalization are non-trivial, even when the problem size is small, due to the high area utilization of packed blocks.

even viable in floorplacement. Indeed, PATOMA and Capo, which pursue correct-by-construction paradigms, appear more robust than FengShui and APlace, which do not attempt to prevent all overlaps in global placement in the interest of improving wirelength.

5. SCALING FLOORPLACEMENT UP

Traditional placement techniques such as top-down and analytical frameworks, bottom-up clustering and iterative cell-spreading, scale well in terms of runtime and interconnect optimization *when all modules are small*. However, handling a wide variety of module sizes with these techniques seems considerably more difficult. On the other hand, simulated annealing has a good track record in handling heterogeneous module configurations, but can only be effectively applied to small problem sizes with our current knowledge.[2] This dichotomy between large-scale placement techniques and annealing-based floorplanning necessitates a rethinking of existing floorplacement flows [24].

In this work we propose a set of mutually-reinforcing floorplacement techniques that significantly improve robustness and performance. As our baseline, we selected the floorplacement framework implemented in the Capo software [2], due to its open-source availability and better-than-average performance in our initial evaluation. However, the enhancements detailed below can also be grafted onto other top-down frameworks, such as those implemented in PATOMA and FengShui. We call our work SCAMPI, an acronym for *SCalable Advanced Macro Placement Improvements*.

Selective floorplanning with macro clustering. In top-down correct-by-construction frameworks like Capo and PATOMA, a key bottleneck is in ensuring ongoing progress — partitioning, floorplanning or end-case processing must succeed at any given step. Both frameworks experience problems when floorplanning is invoked too early to produce reasonable solutions — PATOMA resorts to solutions with very high wirelength, and Capo times out because it has nothing to resort to and runs the Parquet annealer on too many modules. In order to scale better, Parquet clusters small standard cells into soft blocks before starting simulated annealing. When a solution is available, all hard blocks are considered placed and fixed — they are treated as obstacles when the remaining standard cells are placed. Compared to other multi-level frameworks, this one does not include refinement, which makes it relatively fast.

[2]Several attempts at multi-level simulated Annealing, notably Parquet, mPG and MB*-tree, achieved only limited success. However, we feel that this line of research is not exhausted yet.

```
    Variables:  queue of placement partitions
    Initialize queue with top-level partition
1   While (queue not empty)
2     Dequeue a partition
3     If (partition is not marked as merged)
4       Perform look-ahead floorplanning on partition
5       If look-ahead floorplanning fails
6         Undo one partition decision
7         Merge partition with sibling
8         Mark new partition as merged and enqueue
9     Else if (partition has large macros or
          is marked as merged)
10      Mark large macros for placement after floorplanning
11      Cluster remaining macros into soft macros
12      Cluster std-cells into soft macros
13      Use fixed-outline floorplanner to pack
          all macros (soft+hard)
14      If fixed-outline floorplanning succeeds
15        Fix large macros and remove sites beneath
16      Else
17        Undo one partition decision
18        Merge partition with sibling
19        Mark new partition as merged and enqueue
20    Else if (partition is small enough and
          mostly comprised of macros)
21      Process floorplanning on all macros
22    Else if (partition small enough)
23      Process end case std cell placement
24    Else
25      Bi-partition netlist of the partition
26      Divide the partition by placing a cutline
27      Enqueue each child partition
```

Figure 8: Modified min-cut floorplacement (Adya et. al.) flow. Bold-faced lines are new.

Speed is achieved at the cost of not being able to cluster modules other than standard cells because the floorplanner does not produce locations for clustered modules. Unfortunately, this limitation significantly restricts scalability to designs with many macros, as demonstrated earlier on the design *calDCT*.

Our proposed technique of *selective floorplanning with macro clustering* allows to cluster blocks before annealing, and does not require additional refinement or cluster-packing steps (which are among the obvious facilitators) — instead we skip certain existing steps in floorplacement. This improvement is based on two observations: (i) blocks that are much smaller than their bin can be treated like standard cells, (ii) the number of blocks that are large relative to the bin size is necessarily limited. E.g., there cannot be more than nine blocks with area in excess of 10% of a bin's area.

In selective floorplanning, each block is marked as small or large based on a size threshold. Standard cells and small blocks can be clustered, except that clusters containing hard blocks have additional restrictions on their aspect ratios. After successful annealing, only the large blocks are placed, fixed and considered obstacles. Normal top-down partitioning resumes, and each remaining block will qualify as large at some point later. This way, specific locations are determined when the right level of detail is considered. If floorplanning fails during hierarchical placement, we merge the failed bin with its sibling and floorplan the merged bin (see Figure 8). The blocks marked as large in the merged bin include those that exceed the size threshold and also those marked as large in the failed bin (since the failure suggests that those blocks were difficult to pack). After the largest macros are placed, the flow resumes.

The proposed technique limits the size of floorplanning instances given to the annealer *by a constant* (in our case 200 modules) and does not require much extra work. However, it introduces an unexpected complexity. The floorplacement framework implemented in Capo does not handle fixed obstacles in the core region, and none of the public benchmarks have them. When Capo fixes blocks in a particular bin, it fixes *all of them* and never needs to floorplan

```
If block to be added intersects obstacle
  If (block is a left child)
    Find the closest legal location for
      the block to the right of its parent
  Else
    Find the closest legal location for
      the block to the top of its parent
```

Figure 9: Obstacle-evasion during the evaluation of B*-trees.

around obstacles — indeed, Parquet 4.0 does not support fixed obstacles. Another complication due to newly introduced fixed obstacles is in cutline selection. We address both complications below. Of course, reliable obstacle-evasion and intelligent cutline selection may be required by practical designs, even without selective floorplanning (e.g., to handle pre-diffused memories, built-in multipliers in FPGAs, etc). Therefore we view them as independent but synergistic techniques.

Obstacle evasion in floorplanning: B*-tree enhancement. When satisfying area constraints is difficult, it is very important to increase the priority of area optimization so as to achieve legality [10]. Because of this, we select the B*-tree floorplan representation (reviewed in Section 2) over Sequence-Pairs for its amenability to packed configurations.

The original B*-tree paper [8] explains how to handle obstacles by iterating the B*-tree evaluation process so as to avoid overlaps with obstacles. At first, one evaluates a given B*-tree without obstacles, then picks one obstacle and finds the node in the tree closest to the location of the obstacle. The obstacle is then swapped with the node in the tree, using a standard B*-tree move. The tree is re-evaluated, and iterations continue for the remaining obstacles. We found this process to be very slow, and observed that node-swapping moves perturb the initial packing too much, adversely affecting interconnect optimization.

Our new obstacle-evasion algorithm does less work, but accounts for obstacles during the evaluation of the B*-tree, i.e., when the B*-tree is traversed and blocks are successively placed in non-overlapping locations. As each block is added, it is checked for intersection with fixed obstacles. Obstacle-evasion is triggered by any such intersection and alters the B*-tree evaluation process, depending on whether the current block is a left or right child (right children are at the top of their parents, and left children are to the right of their parents). A given block can evade obstacles by moving horizontally or vertically from the location where it would normally be inserted, as shown in Figure 9. In other words, blocks intersecting with fixed obstacles are snapped to the closest legal location consistent with the structure of the B*-tree. Such blocks separate some of their children from the obstacle as well. This change allows one to use the original annealing algorithm without further modifications.

The current implementation runs in $O(N_{bl}N_{ob})$ time, for N_{bl} blocks N_{ob} obstacles because it checks if obstacles overlap with the contour of the B*-tree. In our experience, with intelligent cutline selection and the discreteness of partition boundaries relative to size of floorplanned blocks caused by hierarchical bisection, only a few obstacles need to be accounted for in each partition. A faster implementation of B*-tree contour-obstacle intersection detection may also improve asymptotic complexity, but it may be difficult to improve current empirical performance.

Ad-hoc look-ahead floorplanning. As pointed out in Section 4, the sum of block areas may significantly under-estimate the area required for large blocks. Better estimates are required to improve the robustness of floorplacement as illustrated by Figure 5, and look-ahead area-driven floorplanning appears as a viable approach.

Unlike in PATOMA, where look-ahead is a guarantor of existing solutions, our look-ahead is used as an estimator — more than an oracle and less than a guarantor (since Capo can tolerate failures by backtracking). We use a stronger, less greedy algorithm than those used in PATOMA, and apply it to at most ten blocks at a time. In contrast, PATOMA's look-ahead often processes a large number N_{bl} of blocks in $O(N_{bl} \log N_{bl})$ time, but may overlook many possible solutions.

We perform look-ahead floorplanning to validate solutions produced by the hypergraph partitioner, and check that a resulting partition is packable, within a certain tolerance for failure. Look-ahead floorplanning must be fast, so that the amortized runtime overhead of the look-ahead calls is less than the total time saved from discovering bad partitioning solutions. Therefore look-ahead floorplanning is performed with blocks whose area is larger than 10% of the total module area in the bin, and soft blocks containing remaining modules, except that the size of these soft blocks is artificially reduced. For speed, Parquet is configured to perform area-only packing, and Capo is configured to only perform look-ahead floorplanning on bins with large blocks. Dealing with only the largest blocks is sufficient because floorplanning failures are most often caused by such blocks.

Top-down whitespace allocation by density. The poor quality of area estimates produced by summing block areas also affects whitespace allocation, well-known in standard-cell placement [5]. While uniform whitespace distribution is sufficient in many cases, we observe that in certain cases, one of two child bins requires less whitespace than its sibling. By redistributing whitespace from easy-to-pack child bins to those hard to pack, a floorplacer can become more robust and can also improve runtime.

We propose to adjust whitespace allocation based on *block density*. Given two partitions with equal total block area, we say that a bin with a smaller sum of block-perimeters is *denser*. The sum of perimeters will be greater when there are many blocks, which can be a sign that more whitespace is required. We then alter the traditional uniform whitespace allocation by redistributing whitespace between *sparser* and *denser* bins. This may remind of whitespace allocation to decrease routing congestion based on the concept of *perimeter degree* [25] to estimate whitespace requirements. Since block density is an approximation of the difficulty of floorplanning a bin, we apply our heuristics conservatively in deciding when to redistribute whitespace. For example, when there is a denser child bin with only one macro, it is likely to require less whitespace than its sibling because a single macro has zero dead-space. Conversely, if a dense child bin has a few blocks while its sibling contains significantly more, the denser child bin is likely to benefit from slightly more whitespace, due to the greater amount of dead-space from the packing of large blocks.

6. EXPERIMENTAL RESULTS

Tables 4 and 5 show the performance of our techniques on the proprietary designs and on the IBM-HB$^+$ benchmarks. SCAMPI successfully produces legal placements for all benchmarks and designs. The *calDCT* design consisting of 8827 macros is placed by SCAMPI in under 180s on average, improving upon the scalability of the Capo floorplacement flow which timed-out. Furthermore, the average HPWL of the placement for *calDCT* produced by SCAMPI is 80% better than the best seen solution. The tables also show a significant runtime improvement in SCAMPI over Capo 9.4 on average. This can be attributed to reducing the size of floorplanning windows through clustering, and to better handling of partitioning solutions using look-ahead floorplanning. Figures 1, 2 and 3 plot the placements produced by SCAMPI.

In general, SCAMPI increases the stability, robustness and scalability of the top-down floorplacement framework. We consistently produce legal solutions for all 28 evaluated benchmarks, and achieve best solution quality among all academic tools available to us. In comparison, PATOMA 1.0 and Capo 9.4 were only able to place 18 and 9 of the evaluated benchmarks. Considering the designs and benchmarks successfully placed by PATOMA 1.0 and Capo 9.4, our placements have smaller HPWL by 14.5% and 3.5%, respectively. Our techniques would have been of limited use if they only improved results on the most difficult instances of floorplacement. Therefore we also evaluated SCAMPI against the published performance of Capo on public mixed-size benchmarks from the Faraday and IBM-MSwPins suites [24]. Even though these benchmarks consist mostly of standard cells and have relatively few macros, SCAMPI produces placements with 0.9% lower HPWL and over 10% better runtime.

7. CONCLUSIONS

We described a set of difficult industrial designs that manage to upset published algorithms for floorplacement and academic tools, motivating new research in floorplacement algorithms. We reproduced the problematic behaviors by modifying public-domain netlists, analyzed the performance of published algorithms, identified their limitations, and deduced opportunities for improvement. We then described algorithmic techniques that significantly enhance the scalability and robustness of floorplacement, making it possible for the first time to solve hard instances mentioned above. In particular, obstacle evasion is necessary to support fast multilevel floorplacement of designs with many macros, even when no fixed obstacles are initially present. Our overall results outperform prior state-of-the-art in terms of high success ratios, lower wirelength and lower runtime. As a side-effect, our techniques better distribute whitespace, potentially moderating routing congestion. Furthermore, our improvements are not trade-offs — they enhance the performance of Capo on all benchmark families. As such, the results of this work have been incorporated into Capo 10.0. While the default configuration of Capo 10.0 should be successful on the benchmarks introduced in this paper, reproducing all of our reported results may require running Capo with the option `-SCAMPI`.

We also hope that our work will help improve the quality of other placers. However, our results indicate that analytical placers tend to have difficulties with the type of floorplacement instances we considered.

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

8. REFERENCES

[1] S. N. Adya and I.L. Markov, "Fixed-outline Floorplanning: Enabling Hierarchical Design," *TVLSI* 2003, pp. 1120-35. `http://vlsicad.eecs.umich.edu/BK/parquet/`

[2] S. N. Adya et al., "Unification of Partitioning, Floorplanning and Placement," *ICCAD* 2004, pp. 550-557.

[3] C. J. Alpert, "The ISPD98 Circuit Benchmark Suite," *ISPD* 1998, pp. 80-85.

[4] A. E. Caldwell et al., "Can Recursive Bisection Alone Produce Routable Placements?," *DAC* 2000, pp. 477-482.

[5] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Hierarchical Whitespace Allocation in Top-down Placement," *TCAD* 2003, pp. 716-724.

[6] T. F. Chan et al., "An Enhanced Multilevel Algorithm for Circuit Placement," *ICCAD* 2003, pp. 299-306.

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

[8] Y-C. Chang et al., "B*-trees: A New Representation for Non-Slicing Floorplans," *DAC* 2000, pp. 458-463.

[9] C.-C. Chang, J. Cong and X. Yuan, "Multi-level Placement for Large-Scale Mixed-Size IC Designs," *ASPDAC* 2003, pp. 325-330.

[10] T-C. Chen and Y-W Chang, "Modern Floorplanning based on Fast Simulated Annealing," *ISPD* 2005, pp. 104-112.

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

[12] J. Cong et al., "An Area-Optimality Study of Floorplanning," *ISPD* 2004, pp. 78-83.

[13] J. Cong, M. Romesis and J. Shinnerl, "Fast Floorplanning by Look-Ahead Enabled Recursive Bipartitioning," *ASPDAC* 2005.

[14] J. Cong, M. Romesis and J. Shinnerl, "Robust Mixed-Size Placement Under Tight White-Space Constraints," *ICCAD* 2005.

[15] D. Hill, "Method and System for High Speed Detailed Placement of Cells within an Integrated Circuit Design," US Patent 6370673, April 2002.

[16] A. B. Kahng, "Classical Floorplanning Harmful?," *ISPD* 2000, pp. 207-213.

[17] A. B. Kahng and Q. Wang, "Implementation and Extensibility of an Analytic Placer," *ISPD* 2004, pp. 18-25.

[18] A. B. Kahng and Q. Wang, "An Analytic Placer for Mixed-Size Placement and Timing-Driven Placement," *ICCAD* 2004, pp. 565-572.

[19] A. B. Kahng, S. Reda and Q. Wang, "Architecture and Details of a High Quality, Large-Scale Analytical Placer," *ICCAD* 2005.

[20] G. Karypis et al., "Multilevel Hypergraph Partitioning: Applications in VLSI Design," *DAC* 1997, pp. 526-529.

[21] A. Khatkhate et al., "Recursive Bisection Based Mixed Block Placement," *ISPD* 2004, pp. 84-89.

[22] H. Murata et al., "Rectangle-packing-based module placement," *ICCAD* 1995, pp. 472-479.

[23] J. A. Roy et al., "Capo: Robust and Scalable Open-Source Min-cut Floorplacer," *ISPD* 2005, pp. 224-227.

[24] J. A. Roy et al., 'Min-cut Floorplacement," *to appear in TCAD* 2006.

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

[26] T. Taghavi et al., "Dragon2005: Large-Scale Mixed-size Placement Tool," *ISPD* 2005, pp. 245-247.

[27] N. Viswanathan and C. Chu, "FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model," *ISPD* 2004, pp. 26-33.

[28] K. Vorwerk and A. Kennings, "An Improved Multi-level Framework for Force-Directed Placement," *DATE* 2005, pp. 902-907.

[29] X. Yang, B-K. Choi and M. Sarrafzadeh, "A Standard-Cell Placement Tool for Designs with High Row Utilization," *ICCD* 2002, pp. 45-47.

[30] B. Yao et al., "Unified Quadratic Programming Approach for Mixed Mode Placement," *ISPD* 2005, pp. 193-199.

[31] `http://cadlab.cs.ucla.edu/cpmo/HBsuite.html`

[32] `http://vlsicad.eecs.umich.edu/BK/ISPD06bench`