Constraint-Driven Floorplan Repair

Michael D. Moffitt*, Aaron N. Ng†, Igor L. Markov†, Martha E. Pollack*
*Artificial Intelligence Laboratory / †Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science, University of Michigan
2260 Hayward, Ann Arbor, MI 48109-2121
{mmoffitt,aaronnn,imarkov,pollackm}@eecs.umich.edu

ABSTRACT

Floorplanning algorithms have traditionally underperformed experienced designers, even when relatively simple interconnect metrics are concerned. However, the sheer scale of modern systems on chip makes an all-manual design flow infeasible. In this paper, we propose a new efficient automated approach to the *floorplan repair* problem, where a set of violated design constraints are satisfied by applying small changes to an existing rough floorplan. Such a floorplan can be produced by a human designer, by a scalable placement algorithm, or result from engineering adjustments to a pre-existing floorplan. In all cases, overlapping modules must be separated, and in some instances, modules may need to be repositioned to satisfy other requirements.

The algorithmic framework we propose is built upon an expressive graph-based encoding of constraints. While capable of representing floorplans with or without overlapping modules, it can also support the outline of the core area, fixed module locations, region constraints, proximity and alignment constraints, etc. Instead of applying randomized local search in the hope of satisfying these constraints, we track all implications of imposed constraints and resolve violations by invoking gradual modifications to the floorplan.

The primary focus of this paper is on a particularly efficient *conflict-directed* algorithm for floorplan repair and legalization. It is shown to completely eliminate overlaps from layouts produced by Capo 9.4, Feng Shui 5.1 and APlace 2.01 on IBM-HB benchmarks with hard blocks, typically requiring negligible runtime and increasing interconnect length by only several percent. Furthermore, we are able to generate legal solutions for these instances that surpass previously reported results in wirelength by an average of roughly 7%.

Categories and Subject Descriptors: B.7.2 Integrated Circuits: Design Aids—*placement and routing*; J.6 Computer-Aided Engineering: Computer-Aided Design; G.4 Mathematical Software: Algorithm Design and Analysis

General Terms: Design, Algorithms

Keywords: Floorplanning, Legalization, Constraints

1. INTRODUCTION

The significance and complexity of floorplanning is continually increasing with the growth of systems-on-chip. With hundreds and thousands of modules in modern floorplans, all-manual design is simply too difficult. However, manual floorplanning has remained dominant due to weaknesses of existing algorithms and their inability to efficiently handle important design constraints. Despite improvements in algorithms in both speed and capacity, even the simplest constraint (requiring that modules do not overlap) is often violated by recent floorplanners.

A similar challenge in standard-cell placement has been successfully addressed by decoupling global placement from legalization. The former optimizes interconnect and is allowed to violate many design constraints, which are later enforced by legalization. A great deal of work on legalization in placement has been published [8, 4, 2], but much of it is inapplicable to floorplanning, where most modules have different shapes and do not need to be packed into rows.

In this paper, we propose a new efficient approach to floorplan legalization. The tool we develop – named FLOORIST – performs legalization and repair of existing floorplans, which may have been produced by a variety of layout methodologies. In contrast to some previous attempts at overlap removal in floorplanning [13], our algorithm takes a *conflict-directed* approach to repair, modifying only those features of the layout that are directly responsible for the violated constraints. As a result, the solutions produced by our implementation tend to preserve the qualities and characteristics of the original layout by emulating its initial structure.

2. RELATED WORK

Techniques for ensuring that placed modules do not overlap can be divided into two camps. The first of these includes those strategies that ensure legalization at all steps of search, either by backtracking or by way of look-ahead. The alternative is to allow portions of the layout to remain illegal during search, and afterward perform legalization only as a postprocessing step. Although most existing methods are restricted in scope to standard-cell placement, it is worthwhile to explore how previous studies have attempted to address these issues.

2.1 Correct-by-construction approaches

The mPG package [3] is one instance of a placement tool that enforces legalization at every level of a cluster hierarchy for multiscale optimization. To achieve this, it employs simulated annealing on the sequence-pair representation, a process which can often prove to be expensive. The benefit of this is that each layout produced by mPG is guaranteed to be legal.

Capo [17] is another popular tool which legalizes (or, at least attempts to legalize) subproblems of recursive bipartitioning using simulated annealing. If legalization succeeds, then modules are placed accordingly after some further refinement. If legalization fails, then subproblems are merged, and legalization is attempted on the larger problem. Since no attempt is made to repartition the modules, this failure may in fact propagate to the highest level of search. As a result, Capo can generate infeasible instances if white space is somewhat scarce.

A correct-by-construction framework is also used recently in [4]

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

for mixed-size placement. Here, every subproblem generated is guaranteed to be legalizable, and thus the approach is better characterized as "prelegalization." This is achieved using an extremely simple form of row-based block packing.

2.2 Legalization by post-processing

The mixed-size placer Feng Shui [10] postpones legalization until a complete global floorplan has been generated, making use of a simple Tetris-like algorithm [8]. The analytical engine APlace [9] does the same. In both tools, a greedy algorithm sorts cells by their x-coordinates, and then operates on a cell-by-cell basis, placing each cell into a row that minimizes its total displacement. Macros are handled in a similar fashion.

Other post-placement legalization tools are described in [2] and [16]. In the former, a network flows formulation is constructed to minimize the total (squared) movement of standard cells necessary to achieve legalization. However, the approach is limited, as it cannot modify the position of modules that do not fit into a single cell row. The latter employs diffusion-based legalization, based on a discrete approximation of a continuous diffusion equation; again, cells are presumed to fit into rows.

Perhaps the most relevant work to our problem of post-placement floorplan repair are the approaches proposed in [13] and more recently in [7]. In the former case, a given layout is converted into a sequence-pair formulation, and is then manipulated by perturbing the ordering of pairs of modules and by relocating individual modules. In the latter case, a constraint-graph formulation is used, in which overlaps are removed in a preprocessing step that increases the dimensions of the layout beyond the fixed outline. In both approaches, adjustments are performed to reduce the size of the resulting floorplan. The key difference between these prior efforts and ours is that they do not explicitly encode constraint violations. As a result, the changes made to the layout are not guided to resolve the original violations, nor do the algorithms extend to the repair of anything other than non-overlap constraints.

3. THE FLOORIST ALGORITHM

Our FLOORIST ("*Floorplan Assistant*") legalizer begins with a complete, global floorplan. The violated constraints in this layout may have come from any number of sources, such as flaws induced by the global floorplanner, or the re-sizing of blocks in a previously legal floorplan. This initial solution is then processed by a three-stage procedure. The first stage constructs a pair of constraint graphs from the fixed placements to represent the floorplan. The second stage performs *iterative repair* on these constraint graphs; this is essentially a greedy search, guided by conflicts extracted from critical paths. Finally, the third stage translates the constraint graphs back into a coordinate representation, placing modules as close to their original locations as possible. In this section, we explain these three stages in detail, using a simple example to illustrate our techniques.

3.1 Translation to Constraint Graphs

Once a global floorplanner or chip architect has produced a floorplan, each module has been given a particular location within the fixed outline. In particular, the upper-left corner of any module M_i (having dimensions $w_i \times h_i$) is assigned a coordinate (x_i, y_i) indicating its absolute position. This representation, while expressing the floorplan in its entirety, is not particularly easy to operate upon. Since flaws in the floorplan typically include overlaps between modules, it is more natural to create a structure that reflects pairwise relationships rather than absolute positions.

As a result, we turn to the *constraint graph* representation of a floorplan [11, 15], a formulation that has gained attention in recent studies concerning area-minimization [12]. Here, a pair of graphs

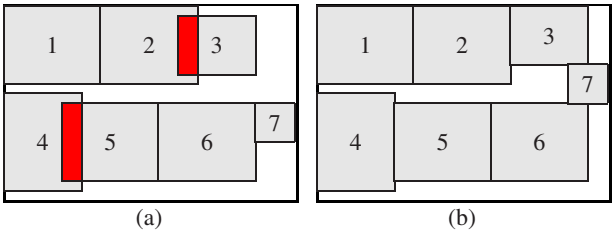


Figure 1: (a) A small floorplan in need of repair. (b) The floorplan after the repair procedure.

(G_H and G_V) is constructed, where each graph contains a node i for every module M_i. Furthermore, there exists a directed edge corresponding to every pair of modules M_i and M_j. The direction of this edge, and the graph that it is placed in, depends on the pairwise relationship between modules M_i and M_j. For instance, suppose that module M_i is to be placed to the left of module M_j, reflecting the condition $x_i + w_i \leq x_j$. This would require an edge from node i to node j in G_H with weight w_i. We will give this pairwise relationship the label $L(i, j)$ in order to facilitate further discussion – the other three possible labels would be $R(i, j)$, $A(i, j)$, $B(i, j)$ (for *right of*, *above*, and *below*). This unified notation allows us to refer to the set S of all pairwise relationships between modules. As an illustration, the following set contains just some of the relationships that could be used to describe the (very small) floorplan shown in Figure 1(a):

$$\{L(1,2), A(1,4), R(3,4), L(5,7)\} \subset S$$

However, notice that modules 2 and 3 overlap in this layout, as do modules 4 and 5. The four relations we have introduced thus far are not sufficient for relating a pair of modules that overlap. Consequently, we propose the addition of a fifth 'empty' relation – $E(i, j)$ – to indicate the lack of any constraint between modules M_i and M_j. The $E(i, j)$ relation can be regarded as the explicit absence of an edge in either constraint graph.

Translation of the existing layout to the constraint graph representation is a fairly straightforward process. Pseudocode for the conversion procedure is given in Figure 2. For every pair of modules that are relatively displaced in a certain direction, the corresponding relation is added to the set S of all pairwise relationships. When multiple relationships are available, ties are broken based on the horizontal and vertical distances between the two modules. For instance, consider modules 1 and 7 in our small example. While edges corresponding to $L(1, 7)$ and $A(1, 7)$ are both feasible choices, we prefer $L(1, 7)$, since the horizontal displacement between these modules is much greater than the vertical displacement. Any pairs of modules that overlap are explicitly given the $E(i, j)$ relation. Once the set S has been constructed, the constraint graphs can be built in $O(N^2)$ time by adding the appropriate edges and computing single-source longest paths.

Create-Graphs$((x_1, y_1, ..., x_N, y_N), (w_1, h_1, ..., w_N, h_N))$
1. $S \leftarrow \emptyset$
2. For $j = 1$ to N
3. For $i = 1$ to j
4. (Of the choices below, keep one w/ smallest slack)
5. If $(x_i + w_i \leq x_j)$ $S \leftarrow S \cup \{L(i, j)\}$
6. If $(x_j + w_j \leq x_i)$ $S \leftarrow S \cup \{R(i, j)\}$
7. If $(y_i + h_i \leq y_j)$ $S \leftarrow S \cup \{A(i, j)\}$
8. If $(y_j + h_j \leq y_i)$ $S \leftarrow S \cup \{B(i, j)\}$
9. (If no choices, $S \leftarrow S \cup \{E(i, j)\}$)
10. return $G_H(S), G_V(S)$

Figure 2: Translating the fixed placement into constraint graphs

Iteratively-Repair(S)

```
1.  While (∃(E(i,j) ∈ S))
2.    // this first loop makes 'trivial' assignments
3.    For each E(i,j) ∈ S
4.      For each possible pairwise relation P(i,j)
5.        If (consistent(S ∪ {P(i,j)} − {E(i,j)}))
6.          S ← S ∪ {P(i,j)} − {E(i,j)}
7.    // this second loop swaps existing assignments
8.    For each E(i,j) ∈ S
9.      For each possible pairwise relation P(i,j)
10.       C ← Critical-Path(M_i) ∪ Critical-Path(M_j)
11.       For each P'(i',j') ∈ C
12.         For each P''(i',j') such that P'' ≠ P'
13.           If (consistent(S ∪ {P''(i',j')} − {P'(i',j')}))
14.             S ← S ∪ {P''(i',j')} − {P'(i',j')}
15.             continue loop @ line 11 with next P'(i',j')
```

Figure 3: Our iterative repair procedure

Of course, constraints other than those extracted from the fixed placements can be added to these graphs. For instance, in most cases there is a fixed outline; to encode this constraint, additional nodes with fixed locations (called "poles") can be added to the graphs representing the walls of the layout, and edges can be imposed between these walls and the modules.

3.2 Conflict-Directed Iterative Repair

For any given set S of pairwise relationships, our objective is to completely remove every empty relation $E(i,j)$ in S, as each of these corresponds to a non-overlap constraint violated in the original layout. To accomplish this task, we would like to make relatively few changes to the original solution, in order to preserve its overall quality. We propose a greedy, backtrack-free iterative repair algorithm, given as pseudocode in Figure 3.

Lines 2 through 6 of this algorithm attempt to replace those $E(i,j)$ relationships whenever one of the four constraining relationships is available. For instance, consider the $E(2,3)$ relation present in our example. We can see that module 3 can be shifted toward the right wall to remove its overlap with module 2. This information can be obtained directly from the constraint graphs, which provide upper and lower bounds on the horizontal positions of modules 2 and 3. As a result, we can instead invoke the relation $L(2,3)$ and update the horizontal constraint graph G_H by adding the constraint $x_2 + w_2 \leq x_3$. The total number of overlaps has now been reduced from two to one.

Unfortunately, the relation $E(4,5)$ cannot be resolved as easily; no other pairwise relationship between modules 4 and 5 can be introduced without either violating some other non-overlap constraint, or extending the layout beyond the given fixed outline. In other words, the edges corresponding to the relations in the set $\{L(4,5), R(4,5), A(4,5), B(4,5)\}$ are in *conflict* with the current constraint graphs, and hence we must perform some modifications before we can repair this particular violated constraint.

Again, by inspection, we can identify some possible alternatives by examining Figure 1(a). In particular, we could place module 7 above module 6, rather than to its right. This would amount to removing the relation $L(6,7)$ (and its corresponding edge in G_H) and replacing it with $B(6,7)$ and the constraint $y_7 + h_7 \leq y_6$ (adding the appropriate edge in G_V). After performing this modification, the last remaining overlap can be resolved by adding the relation $L(4,5)$ and the constraint $x_4 + w_4 \leq x_5$. There exist other relations that can be swapped in this example, but few would help to resolve our violated constraint. For instance, we could reverse the relationship between modules 1 and 2 by replacing $L(1,2)$ with $R(1,2)$, but such a modification would have no effect on the overlap between modules 4 and 5.

This scenario alludes to a *conflict-directed* approach to legalization, and is the key concept in our approach to floorplan repair. Specifically, for each of the four primary relations of a violated non-overlap constraint, we can identify a set of *culprits* C leading to its inapplicability by examining the edges along the critical paths[1] for modules M_i and M_j in the appropriate constraint graph (lines 8 through 10 in Figure 3). If any of these edges can be safely replaced with an alternative pairwise relationship (lines 11 through 13), then we perform this replacement (line 14).[2] Each swap can be achieved in $O(N^2)$ time using the longest-path algorithm (although performing incremental maintenance of the graph when adding edges can help to improve efficiency). In the event that some previously violated constraints can now be satisfied, these will be caught during the first phase of the next step in iterative repair. This process continues indefinitely until all violated constraints have been resolved.

3.3 Translation to Fixed Locations

The final stage in our legalization procedure is the recreation of a fixed placement solution. Recall that since the edges in our constraint graphs record only the relative positions of pairs of modules, any layout extracted from these graphs may differ significantly from the initial solution. In the event that some modules exhibit a great deal of slack, one would prefer to place such modules as close to their original positions as possible, in order to preserve qualities (such as wirelength) that the global floorplanner presumably extended a considerable amount of effort to optimize.

The pseudocode given in Figure 4 shows how we perform this emulation. For each module, we use the horizontal constraint graph G_H to extract the lower and upper bounds on its final horizontal position x_i. From this, we can determine whether or not it can be given its original horizontal coordinate ($orig\_x_i$). If it can, then this is the assignment it receives. Otherwise, the module is slid as far to the left or to the right as possible, depending on which side of its original position it lies. A similar operation is performed to determine the module's vertical position. These fixed assignments are propagated through the constraint graphs, and the next module is processed. Since each call to **Update-Constraint-Graphs** requires $O(N^2)$ time, the entire emulation procedure is $O(N^3)$.[3] Although the modules can be examined in any order, we recommend beginning with the largest modules first, since these will have a greater impact on the final layout than smaller ones. The output of the combined legalization and emulation process for our running example is shown in Figure 1(b).

[1] As explained in [1], a *critical path* is a path of blocks that constrain each other in the same direction and are tightly packed.

[2] We make use of the modules' original locations to choose amongst the viable relations; more intelligent heuristics may lead to further improvements.

[3] While this process may seem expensive, it occurs only once, and can be sped up if preserving initial locations is not important. In addition, while $\Theta(N^3)$-time complexity is usually considered prohibitive for layout algorithms, we note that sequence-pair annealers are typically slower, performing N moves at each temperature value and at least N^2 operations at each move.

Create-Fixed-Placement(G_H, G_V)

```
1.  For i = 1 to N
2.    If (G_H.lb(x_i) > orig_x_i) x_i ← G_H.lb(x_i)
3.    Else If (G_H.ub(x_i) < orig_x_i) x_i ← G_H.ub(x_i)
4.    Else x_i ← orig_x_i
5.    If (G_V.lb(y_i) > orig_y_i) y_i ← G_V.lb(y_i)
6.    Else If (G_V.ub(y_i) < orig_y_i) y_i ← G_V.ub(y_i)
7.    Else y_i ← orig_y_i
8.    Update-Constraint-Graphs(G_H,G_V)
```

Figure 4: The creation of a fixed placement from the constraint graphs

Circuit	Macros	Nets	Previous Best HPWL [5]	APlace + FLOORIST HPWL	
ibm-HB01	911	5829	3.10 e+06	2.86 e+06	–7.6%
ibm-HB02	1471	8508	6.42 e+06	6.37 e+06	–1.0%
ibm-HB03	1289	10279	9.80 e+06	8.98 e+06	–8.4%
ibm-HB04	1584	12456	11.0 e+06	9.87 e+06	–10.%
ibm-HB05	564	9171	14.6 e+06	13.8 e+06	–5.4%
ibm-HB06	749	9963	8.83 e+06	8.23 e+06	–6.8%
ibm-HB07	1120	15047	17.0 e+06	15.9 e+06	–6.4%
ibm-HB08	1269	16075	18.8 e+06	16.8 e+06	–10.%
ibm-HB09	1113	18913	18.7 e+06	18.4 e+06	–1.3%
ibm-HB10	1595	27508	53.9 e+06	n/a	n/a
ibm-HB11	1497	27477	28.9 e+06	27.8 e+06	–3.7%
ibm-HB12	1233	26320	58.6 e+06	51.6 e+06	–12.%
ibm-HB13	954	27011	36.9 e+06	35.4 e+06	–4.1%
ibm-HB14	1635	43062	66.0 e+06	60.0 e+06	–9.1%
ibm-HB15	1412	52779	90.4 e+06	82.5 e+06	–8.7%
ibm-HB16	1091	47821	103. e+06	91.3 e+06	–11.%
ibm-HB17	1442	56517	146. e+06	138. e+06	–5.6%
ibm-HB18	943	42200	73.2 e+06	68.7 e+06	–6.1%
				Average Reduction	**–6.9%**

Table 1: The IBM-HB benchmarks

3.4 Repairing Other Constraint Types

As mentioned in [18], non-overlap constraints are just one of many types of constraints that traditional constraint graphs can express. For instance, region constraints, proximity constraints, and alignment constraints can all be represented by edges in the graph, and similarly, their violation can be regarded as the *absence* of such edges. Consequently, our conflict-directed approach can repair these constraints just as easily by manipulating the critical paths that render them infeasible.

To illustrate this ability, we present a series of images in Figure 5, showing how FLOORIST repairs a violated region constraint. The initial solution is displayed in the leftmost layout; the green shaded block currently violates a constraint that requires its center to be aligned with the horizontal midsection of the floorplan. Second, we show the movement of blocks as displacement vectors, to highlight how the majority of the modules are largely unaffected by this repair. In the third image we show the final layout after FLOORIST has repaired this constraint.

4. EXPERIMENTAL RESULTS

In order to evaluate the efficacy of our constraint-driven approach to floorplan repair, we ran three global floorplanners – Capo 9.4 [17], Feng Shui 5.1 [10], and APlace 2.01 [9] – on the IBM-HB benchmarks described in [5]. A summary of these benchmarks, along with the wirelength of the best known solutions, is provided in Table 1 for reference (the final two columns will be discussed later). Since the latter two tools do not support soft blocks, we convert all soft blocks to hard blocks with an aspect ratio of 1.0. For both Feng Shui and APlace we used the default amount of 20% whitespace, which consistently produced layouts with overlapping modules. However, Capo often solved these instances without need for legalization, and so we reduce whitespace to 15% for its instances in order to generate solutions that do contain overlapping modules. Importantly, this means that the quality of its initial placements (particularly with respect to wirelength) are incomparable to the other floorplanning packages, especially since Capo's performance improves significantly with increased whitespace.[4]

We then ran three legalizers on the output produced by the global floorplanners. The first is part of the Feng Shui package, and (as mentioned earlier) is a variation on the Tetris [8] algorithm.[5] The second legalizer is Parquet4.5 [1], an annealer used in Capo for solving end-cases in subproblem recursion. Although it was not originally intended for legalization, it can be used for this purpose by setting the annealing temperature at a relatively low value, and establishing a termination criterion in area-minimization mode so that it stops whenever its solution can fit within the outline. The final legalizer is our FLOORIST algorithm. A timeout of 120 seconds was enforced on all problems.

Table 2 displays the results of these experiments. The table is divided into three sections, one for each global floorplanner. The leftmost four columns of the table give the name of the instance and report the initial solutions produced by the global floorplanner, displaying overlap as a percentage of layout area, the half-perimeter wirelength, and runtime. The remaining columns give similar statistics for each of the three legalizers. We also report the relative increase in wirelength of the legalized solutions over the original floorplan. Since overlapping modules contribute to shorter wirelength, a high percentage may indicate a particularly illegal initial solution, and thus one should use these percentages only to compare legalizers to each other, and to compare initial placements to each other. We have also plotted some of the layouts produced by the global floorplanners and legalizers in Figure 6. For each legalized solution, we have added displacement vectors, showing how each module has moved from its original location.

Of all three legalizers, we find that the tool provided in the Feng Shui 5.1 release is by far the most unreliable, crashing on the majority of instances. Furthermore, in those cases where it does generate a solution, it often *increases* the amount of overlap (especially for Capo's solutions, and some of APlace's layouts). It also tends to violate the fixed outline constraint, placing modules out of core. In fact, there are no instances where it successfully legalizes the layout completely.

Parquet, on the other hand, removes overlaps from all floorplans, and does this rather quickly in many cases. However, on some of Capo's and Feng Shui's initial solutions, it cannot satisfy the fixed outline constraint within the time limit, and so these solutions are not legal. In addition, the wirelengths of its solutions are often twice or three times as large as those of the initial placement. This may be due to the fact that it operates on a sequence-pair formulation, in which a small perturbation may cause a module to be moved significantly from its original position.

In contrast, FLOORIST achieves legality and 0% overlap on all instances, preserving the length of most individual wires (and thus global interconnect length) far better than the other legalizers while remaining competitive in runtime. For Capo's layouts, it typically requires roughly 0% increase in wirelength; this is largely due to the fact that the overlaps in Capo's layouts are often very small, occuring only faintly on the border of neighboring modules. On the layouts produced by Feng Shui's global floorplanner, the increase in wirelength is more pronounced; however, the blame for this can likely be attributed to the considerably high amount of overlap present in Feng Shui's initial placements. Finally, we find that the results produced by using FLOORIST on APlace's initial solutions are each superior in wirelength to the best legal solutions known previously for these instances. As shown in Table 1, the average reduction in wirelength is calculated to be roughly 7%. What makes these results particularly interesting is that we are working with hard blocks having fixed square dimensions, whereas the previous solutions were obtained by allowing blocks to be soft.

5. CONCLUSION

In this paper, we have developed a new efficient approach to post-placement floorplan repair. Our FLOORIST algorithm legalizes existing floorplans using constraint-driven, conflict-directed mod-

[4]The recent release of Capo10 [14] performs much better than Capo9.4; however, we use the older version because (a) Capo10 places the IBM-HB variants with 15% whitespace legally without overlaps, (b) while Floorist succeeded on all variants with 10% whitespace, Feng Shui's legalizer failed in all cases.

[5]We have decoupled the global floorplanner in Feng Shui 5.1 from its standalone legalizer in our experiments, in order to evaluate them independently.

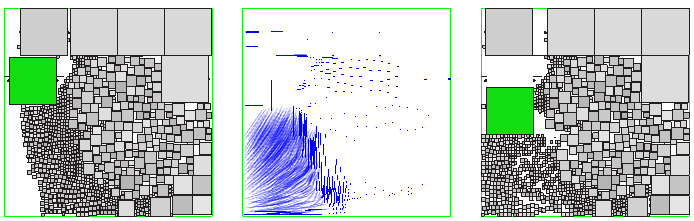
Table 2: Performance of legalizers on output from Capo 9.4, Feng Shui 5.1, and APlace 2.01

white-space 15%	Capo 9.4 Solution ovlp (%)	Capo 9.4 Solution HPWL (e+06)	Capo 9.4 Solution time (sec.)	FS 5.1's "Tetris" legalizer ovlp (%)	FS 5.1's "Tetris" legalizer HPWL (e+06)	FS 5.1's "Tetris" legalizer leg.time (sec.)	Parquet4.5 ovlp (%)	Parquet4.5 HPWL (e+06)	Parquet4.5 leg.time (sec.)	Floorist (our work) ovlp (%)	Floorist (our work) HPWL (e+06)	Floorist (our work) leg.time (sec.)
HB01	0.16	3.57	988	**0.891oc**	4.01 (+12%)	15.9	**0.00**	4.14 (+16%)	6.17	**0.00**	3.57 (+0.0%)	1.89
HB02	0.21	16.5	2659	– – – *seg fault* – – –			**0.00oc**	21.1 (+27%)	120.	**0.00**	16.5 (+0.0%)	9.91
HB03	0.07	17.1	2404	– – – *seg fault* – – –			**0.00**	19.5 (+14%)	11.3	**0.00**	17.1 (+0.0%)	4.51
HB04	0.17	11.6	3404	– – – *seg fault* – – –			**0.00**	13.3 (+15%)	13.0	**0.00**	11.6 (+0.0%)	10.9
HB05	0.24	14.6	757	– – – *seg fault* – – –			**0.00**	14.6 (+ 0%)	3.35	**0.00**	14.6 (+0.0%)	0.56
HB06	0.13	9.32	462	– – – *seg fault* – – –			**0.00**	11.4 (+22%)	5.12	**0.00**	9.32 (+0.0%)	1.59
HB07	0.16	17.8	577	**0.873oc**	19.2 (+18%)	7.02	**0.00**	17.7 (– 1%)	8.52	**0.00**	17.9 (+0.6%)	5.22
HB08	0.12	22.6	1257	– – – *seg fault* – – –			**0.00**	23.6 (+ 4%)	9.75	**0.00**	22.6 (+0.0%)	5.61
HB09	0.20	35.8	2684	– – – *seg fault* – – –			**0.00oc**	47.1 (+32%)	120.	**0.00**	35.8 (+0.0%)	4.71
HB10	0.26	72.9	5185	– – – *seg fault* – – –			**0.00oc**	117. (+60%)	120.	**0.00**	74.0 (+1.4%)	20.9
HB11	0.06	69.4	6479	– – – *seg fault* – – –			**0.00oc**	77.5 (+12%)	120.	**0.00**	69.4 (+0.0%)	6.28
HB12	0.03	93.1	4408	– – – *seg fault* – – –			**0.00oc**	102. (+ 9%)	120.	**0.00**	93.1 (+0.0%)	2.71
HB13	0.08	45.6	960	– – – *seg fault* – – –			**0.00**	52.5 (+15%)	7.02	**0.00**	45.6 (+0.0%)	2.27
HB14	0.13	65.6	1329	– – – *seg fault* – – –			**0.00**	65.0 (– 1%)	14.7	**0.00**	65.6 (+0.0%)	10.9
HB15	0.09	98.1	5752	– – – *seg fault* – – –			**0.00oc**	116. (+18%)	120.	**0.00**	98.1 (+0.0%)	7.59
HB16	0.12	149.	3690	– – – *seg fault* – – –			**0.00**	159. (+ 7%)	18.1	**0.00**	149. (+0.0%)	2.78
HB17	0.07	153.	2149	– – – *seg fault* – – –			**0.00oc**	176. (+15%)	120.	**0.00**	153. (+0.0%)	5.94
HB18	0.10	74.2	1390	**5.696oc**	88.3 (+19%)	90.6	**0.00oc**	119. (+60%)	120.	**0.00**	74.2 (+0.0%)	2.48
Avg.	0.12	53.9	2585	Insufficient Data			**0.00oc**	**+ 18.1%**	**58.7**	**0.00**	**+ 0.1%**	**5.93**

white-space 20%	FS 5.1 w/o legalization ovlp (%)	FS 5.1 w/o legalization HPWL (e+06)	FS 5.1 w/o legalization time (sec.)	FS 5.1's "Tetris" legalizer ovlp (%)	FS 5.1's "Tetris" legalizer HPWL (e+06)	FS 5.1's "Tetris" legalizer leg.time (sec.)	Parquet4.5 ovlp (%)	Parquet4.5 HPWL (e+06)	Parquet4.5 leg.time (sec.)	Floorist (our work) ovlp (%)	Floorist (our work) HPWL (e+06)	Floorist (our work) leg.time (sec.)
HB01	10.2oc	2.94	20.8	**0.84oc**	3.76 (+28%)	3.41	**0.00**	9.04 (+207%)	35.8	**0.00**	3.21 (+ 9%)	11.9
HB02	– – – *crashes* – – –			n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HB03	7.98oc	9.05	39.9	– – – *seg fault* – – –			**0.00**	27.3 (+202%)	58.3	**0.00**	10.1 (+12%)	65.1
HB04	7.09oc	10.4	48.6	– – – *seg fault* – – –			**0.00**	31.9 (+208%)	48.8	**0.00**	11.5 (+11%)	49.4
HB05	– – – *crashes* – – –			n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HB06	6.23oc	8.16	36.0	– – – *seg fault* – – –			**0.00**	28.0 (+243%)	9.16	**0.00**	8.85 (+ 8%)	20.2
HB07	– – – *crashes* – – –			n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HB08	7.84oc	17.6	62.3	– – – *seg fault* – – –			**0.00**	55.0 (+213%)	34.7	**0.00**	19.4 (+10%)	27.3
HB09	6.53oc	17.2	47.5	**1.84oc**	19.2 (+11%)	6.42	**0.00oc**	76.2 (+342%)	120.	**0.00**	18.8 (+ 9%)	14.3
HB10	– – – *crashes* – – –			n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HB11	7.90oc	27.7	69.1	**1.64oc**	32.0 (+15%)	10.4	**0.00**	95.0 (+243%)	51.0	**0.00**	29.9 (+ 8%)	45.0
HB12	4.81oc	54.9	48.5	– – – *seg fault* – – –			**0.00**	88.0 (+ 60%)	9.80	**0.00**	59.3 (+ 8%)	9.16
HB13	7.11oc	37.3	57.1	**0.183oc**	45.5 (+22%)	3.62	**0.00**	100. (+169%)	23.6	**0.00**	40.4 (+ 8%)	10.2
HB14	6.67oc	60.2	110.	– – – *seg fault* – – –			**0.00oc**	155. (+157%)	120.	**0.00**	65.0 (+ 8%)	58.5
HB15	5.71oc	82.8	117.	**2.169oc**	103. (+24%)	5.61	**0.00**	221. (+166%)	68.7	**0.00**	89.5 (+ 8%)	40.1
HB16	6.36oc	97.1	84.2	**0.004oc**	108. (+11%)	3.89	**0.00**	265. (+173%)	24.9	**0.00**	106. (+ 9%)	16.2
HB17	8.39oc	140.	118.	– – – *seg fault* – – –			**0.00oc**	263. (+ 88%)	120.	**0.00**	152. (+ 8%)	81.7
HB18	7.20oc	69.3	90.1	– – – *seg fault* – – –			**0.00**	169. (+144%)	35.7	**0.00**	75.1 (+ 8%)	22.1
Avg.	7.14oc	39.77	67.8	Insufficient Data			**0.00oc**	**+ 187%**	**54.3**	**0.00**	**+ 8.9%**	**32.8**

white-space 20%	APlace 2.01 Solution ovlp (%)	APlace 2.01 Solution HPWL (e+06)	APlace 2.01 Solution time (sec.)	FS 5.1's "Tetris" legalizer ovlp (%)	FS 5.1's "Tetris" legalizer HPWL (e+06)	FS 5.1's "Tetris" legalizer leg.time (sec.)	Parquet4.5 ovlp (%)	Parquet4.5 HPWL (e+06)	Parquet4.5 leg.time (sec.)	Floorist (our work) ovlp (%)	Floorist (our work) HPWL (e+06)	Floorist (our work) leg.time (sec.)
HB01	2.73	2.66	66.83	**0.931**	3.18 (+20%)	2.30	**0.00**	6.32 (+138%)	10.6	**0.00**	2.86 (+ 8%) ✓	4.67
HB02	2.62	5.63	126.5	– – – *seg fault* – – –			**0.00**	17.9 (+218%)	17.2	**0.00**	6.37 (+13%) ✓	44.1
HB03	2.13	8.18	130.5	– – – *seg fault* – – –			**0.00**	16.8 (+105%)	10.6	**0.00**	8.98 (+10%) ✓	13.4
HB04	2.84	9.08	127.2	– – – *seg fault* – – –			**0.00**	23.1 (+154%)	18.9	**0.00**	9.87 (+ 9%) ✓	28.9
HB05	0.549oc	13.8	502.1	– – – *seg fault* – – –			**0.00**	14.5 (+ 5%)	3.64	**0.00**	13.8 (+ 0%) ✓	0.89
HB06	1.64	7.94	94.39	– – – *seg fault* – – –			**0.00**	15.5 (+ 95%)	5.36	**0.00**	8.23 (+ 4%) ✓	2.49
HB07	1.37	15.4	314.1	**7.996oc**	17.4 (+13%)	12.2	**0.00**	30.4 (+ 98%)	9.00	**0.00**	15.9 (+ 4%) ✓	7.97
HB08	1.13oc	16.5	340.3	**1.728oc**	19.2 (+16%)	19.5	**0.00**	31.0 (+ 87%)	10.9	**0.00**	16.8 (+ 2%) ✓	9.38
HB09	1.13oc	17.7	222.3	– – – *seg fault* – – –			**0.00**	42.6 (+141%)	9.03	**0.00**	18.4 (+ 4%) ✓	7.54
HB10	– – – *program hangs* – – –			n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
HB11	1.02	27.4	444.2	– – – *seg fault* – – –			**0.00**	52.9 (+ 93%)	14.21	**0.00**	27.8 (+ 1%) ✓	14.7
HB12	0.19	51.6	301.1	**0.001oc**	54.9 (+ 7%)	4.49	**0.00**	71.3 (+ 38%)	11.2	**0.00**	51.6 (+ 0%) ✓	3.37
HB13	0.55	35.2	276.7	– – – *seg fault* – – –			**0.00**	53.1 (+ 51%)	7.64	**0.00**	35.4 (+ 1%) ✓	2.25
HB14	1.07	58.5	542.6	– – – *seg fault* – – –			**0.00**	94.1 (+ 61%)	16.8	**0.00**	60.0 (+ 3%) ✓	18.0
HB15	0.78	81.6	633.9	– – – *seg fault* – – –			**0.00**	117. (+ 43%)	13.7	**0.00**	82.5 (+ 1%) ✓	11.7
HB16	0.34	91.2	736.5	– – – *seg fault* – – –			**0.00**	128. (+ 40%)	8.75	**0.00**	91.3 (+ 0%) ✓	4.37
HB17	0.53oc	138.	938.2	**0.022oc**	150. (+ 9%)	13.4	**0.00**	181. (+ 32%)	13.4	**0.00**	138. (+ 0%) ✓	5.09
HB18	0.57	68.0	456.7	– – – *seg fault* – – –			**0.00**	111. (+ 64%)	8.52	**0.00**	68.7 (+ 1%) ✓	3.15
Avg.	1.25	38.1	367.9	Insufficient Data			**0.00**	**+ 86%**	**11.1**	**0.00**	**+ 3.5%**	**10.7**

A ✓ indicates a new best known legal solution. An 'oc' indicates an out-of-core solution.



(a) Initial (b) Movement (c) Final

Figure 5: Imposing a region constraint on the green block in the above HB12 instance forces it (and blocks beneath it) to move down.

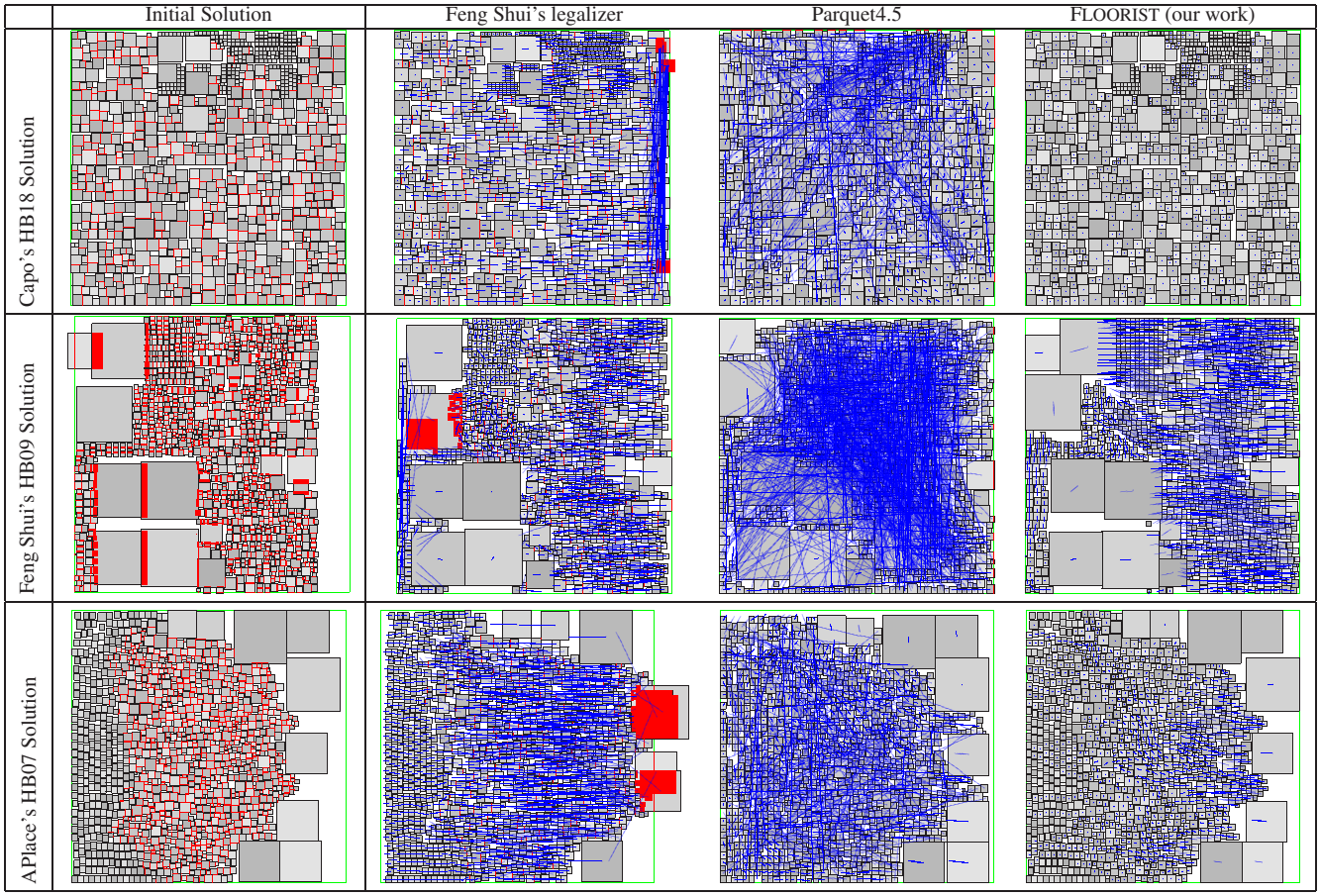


Figure 6: Initial Solutions and Outputs of Legalizers (Red regions indicate overlap and violations of the fixed outline constraint; blue vectors indicate movement of modules). All initial solutions have numerous overlaps. Feng Shui's legalizer produces solutions that contain overlapping modules and modules out-of-core. Parquet removes all overlaps, but two solutions violate the outline constraint, and all placements require extensive movement of modules. FLOORIST's solutions are entirely legal, and require relatively little module movement.

ifications. In contrast to previous attempts at overlap removal in floorplanning, our algorithm modifies only those features of the layout that are directly responsible for the violated constraints, and thus preserves the qualities and characteristics of the original layout by emulating its initial structure. The versatility of this approach makes it useful for a number of applications, allowing one to post-process the outputs of global floorplanners, fix overlaps introduced by the re-sizing of modules in existing floorplans, and to legalize rough floorplans sketched by chip architects.

6. REFERENCES

[1] S. N. Adya and I. L. Markov. Fixed-outline Floorplanning: Enabling Hierarchical Design. In *IEEE Trans. on VLSI Systems*, vol. 11(6), pages 1120–1135, 2003.

[2] U. Brenner, A. Pauli, and J. Vygen. Almost optimum placement legalization by minimum cost flow and dynamic programming. In *Proc. of ISPD '04*, pages 2–9, 2004.

[3] C. Chang, J. Cong, and X. Yuan. Multi-level placement for large-scale mixed-size ic designs. In *Proc. of ASP-DAC '03*, pages 325–330, 2003.

[4] J. Cong, M. Romesis, and J. R. Shinnerl. Robust mixed-size placement under tight white-space constraints. In *Proc. of ICCAD '05*, pages 165–172, 2005.

[5] J. Cong, M. Romesis, and J. Shinnerl. Fast floorplanning by look-ahead enabled recursive bipartitioning. Technical Report TR040043, Computer Science Dept., UCLA, 2004.

[6] J. Cong, M. Romesis, and J. Shinnerl. Fast floorplanning by look-ahead enabled recursive bipartitioning. In *Proc. of ASP-DAC '05*, pages 1119–1122, 2005.

[7] J. Cong and M. Xie. A Robust Detailed Placement for Mixed-Size IC Designs. In *Proc. of ASP-DAC '06*, pages 188–194, 2006.

[8] D. Hill. Method and system for high speed detailed placement of cells within an integrated circuit design, US Patent 6370673, April 2002.

[9] A. B. Kahng, S. Reda, and Q. Wang. Architecture and details of a high quality, large-scale analytical placer. In *Proc. of ICCAD '05*, 2005.

[10] A. Khatkhate, et. al. Recursive bisection based mixed block placement. In *Proc. of ISPD '04*, pages 84–89, 2004.

[11] Y. Liao and C. K. Wong. An algorithm to compact a VLSI symbolic layout with mixed constraints. In *IEEE Transactions on CAD, Vol. 2, No. 2*, 1983.

[12] M. D. Moffitt and M. E. Pollack. Optimal rectangle packing: a meta-CSP approach. To appear in *Proc. of ICAPS '06*, 2006.

[13] S. Nag and K. Chaudhary. Post-Placement Residual-Overlap Removal with Minimal Movement. In *Proc. of DATE '99*, pages 581–586, 1999.

[14] A. N. Ng, I. L. Markov, R. Aggarwal and V. Ramachandran. Solving Hard Instances of Floorplacement. In *Proc. of ISPD '06*, pages 170–177, 2006.

[15] H. Onodera, Y. Taniguchi, and K. Tamaru. Branch-and-bound placement for building block layout. In *Proc. of DAC '91*, pages 433–439, 1991.

[16] H. Ren, D. Z. Pan, C. J. Alpert, and P. Villarrubia. Diffusion-based placement migration. In *Proc. of DAC '05*, pages 515–520, 2005.

[17] J. A. Roy, et. al. Capo: robust and scalable open-source min-cut floorplacer. In *Proc. of ISPD '05*, pages 224–226, 2005.

[18] E. Young, M.L. Ho, and C. Chu. A Unified Method to Handle Different Kinds of Placement Constraints in Floorplan Design. In *Proc. of ASP-DAC '02*, pages 661–670, 2002.