Capo: Congestion-driven Placement for Standard-cell and RTL Netlists with Incremental Capability

Jarrod A. Roy, David A. Papa and Igor L. Markov

The University of Michigan, Department of EECS
2260 Hayward Ave., Ann Arbor, MI 48109-2121
{royj,iamyou,imarkov}@eecs.umich.edu

Summary. In this chapter, we describe the robust and scalable academic placement tool Capo. Capo uses the min-cut placement paradigm and performs (i) scalable multi-way partitioning, (ii) routable standard-cell placement, (iii) integrated mixed-size placement, (iv) wirelength-driven fixed-outline floorplanning as well as (v) incremental placement.

1 Introduction

The success of min-cut techniques in fixed-die placement is based on the speed and strength of multi-level hypergraph partitioners, the convenient top-down framework that efficiently captures available on-chip resources, and the fact that modern VLSI circuits admit a large number of good placements, which include slicing placements. The recent trend for large amounts of whitespace, clearly visible in the ISPD05 and ISPD06 contest benchmarks, particularly increases the flexibility in the placement problem.

The earliest work describing the Capo placer was a paper from ISPD 1999 describing the end-case placers and optimal partitioners as well as terminal propagation with inessential nets used in Capo [13]. The Capo placer, first released at DAC 2000 [11], sought to produce routable placements with a pure min-cut algorithm. To this end, Capo 8.0 was successful for most industrial benchmarks evaluated, even though it did not build or use congestion maps. For example, it produced a routable placement of an industrial design with 200K cells in 1.5 hours on a single-processor workstation. Capo's routability was evaluated with a full-fledged router and demonstrated that early estimators of routability may produce misleading results [11].

Capo's overall performance was on par with commercial tools, however an ISPD 2002 paper [41] proposed a new set of benchmarks on which Capo was less successful compared to a newer tool, Dragon. Dragon found routable placements in most cases by building congestion maps and biasing the placement process accordingly. This suggested that congestion-driven placement was far from solved and several papers in 2003-2005 and later reported even better results [1, 5, 23, 27].

Earlier versions of Capo distributed whitespace approximately uniformly, according to the hierarchical whitespace distribution formula from [15]. However more recent work [4] introduces tunable whitespace distribution for improved wirelength, while preserving a minimum amount of local whitespace in most regions to ensure routability. Whitespace allocation and

detail placement have been further improved by analyzing the performance of Capo on *feature benchmarks* [32] designed to stress different aspects of placers.

Unlike Dragon and FengShui [5], Capo does not explicitly use multi-way partitioning. The addition of *placement feedback* [24] counteracts this potential limitation. Additionally, cutline shifting in recursive bisection adds flexibility in partition shapes and sizes, as well as whitespace allocation; this is not readily available in direct min-cut multi-way partitioning.

The most recent work on Capo has been on improving Capo's performance on routing benchmarks and difficult instances of floorplanning and mixed-size placement, and transforming Capo into an incremental placement tool. As of Fall 2006, Capo produces the best published routed wirelengths on several suites of routing benchmarks by directly optimizing Steiner wirelength and cut-line shifting based on congestion [35]. Capo also performs efficiently with good solution quality on difficult instances of floorplacement which are not legally placeable by several other academic techniques [31]. Incremental placement in Capo consists of simulating the decisions a min-cut placer may have made to produce a given initial placement [36]. For each decision that is made, Capo chooses to accept or reject the decision. Accepting a particular decision means continuing the simulation of decisions whereas rejecting a decision results in replacement of a part of the design from scratch. Empirical results show that Capo's incremental placement moves objects minimally, produces solutions with good HPWL, and runs faster than other available legalization techniques [36].

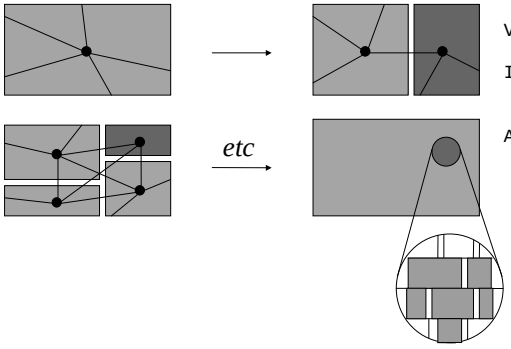
Using the min-cut floorplacement algorithm from [34] and improvements introduced in [31, 35, 36], Capo 10 performs (i) scalable multi-way partitioning, (ii) routable standard-cell placement, (iii) integrated mixed-size placement, (iv) wirelength-driven fixed-outline floorplanning and (v) incremental placement. Capo was used by Synplicity in the Amplify ASIC product. In particular, Amplify ASIC RC targeted LSI Logic's RapidChip architecture. Most RapidChip designs produced were placed with Capo, and successful customers include companies such as HP, SGI, CISCO, Nortel Networks, Raytheon, Seagate, 3COM, Alcatel, Hitachi, Fujitsu, IP Wireless, Cryptek, etc. Source code and executables of Capo 10 are available at `http://vlsicad.eecs.umich.edu/BK/PDtools/`.

2 Min-cut Placement in Capo

Row-Based Placement. Internally, Capo's placement representation closely resembles the LEF/DEF and Bookshelf [14] file formats, which represent row information in standard-cell layout. Configurations of rows supply constraints for cell placement. Each row consists of non-overlapping subrows aligned to the coordinate of the row. All subrows in a row share the same coordinate, height, site width and site spacing. Placement instances in the Bookshelf format consist of several rows composed of one or more subrows.

Fixed objects may displace sites in the core region. Since fixed objects prevent standard cells from being placed in those sites, they are *obstacles*. To prevent the placer from using sites occupied by obstacles, one solution is to remove the sites beneath all fixed objects. Capo accomplishes this by fracturing the rows containing the occupied sites into subrows, excluding the sites beneath the obstacle [11, Sec. 4.2]. The result is a row-based placement structure containing only legal locations for placing standard cells.

Min-Cut Bisection. Top-down placement algorithms seek to decompose a given placement instance into smaller instances by sub-dividing the placement region, assigning modules to subregions and cutting the netlist hypergraph [11] (see Figure 1). Min-cut placers generally use either bisection or quadrisection to divide the placement area and netlist. Capo uses bisection




```
Variables:
    A queue of blocks
Initialization:
    A single block represents
    the original placement problem
Algorithm:
    while (queue not empty)
    dequeue a block
    if (small enough) consider endcase
    else {
    bipartition into smaller blocks
    enqueue each block
    }
```

Fig. 1. High-level outline of the top-down partitioning-based placement process [13]. ©2000 IEEE.

as it allows for greater flexibility in cutline shifting to adapt to changing partition sizes [11, Sec. 3.2].

Each hypergraph partitioning instance is induced from a rectangular region, or bin, in the layout. In this context a *placement bin* represents (i) a placement region with allowed module locations (*sites*), (ii) a collection of circuit modules to be placed in this region, (iii) all signal nets incident to the modules in the region, and (iv) fixed cells and pins outside the region that are adjacent to modules in the region (*terminals*). Top-down placement can be viewed as a sequence of passes where each pass examines all bins and divides some of them into smaller bins.

Capo implements three types of min-cut partitioners – optimal (branch-and-bound [13]), middle-range (Fiduccia-Mattheyses [12]) and large-scale (multi-level Fiduccia-Mattheyses partitioner MLPart [10]). Bins with seven or fewer cells use an optimal end-case placer. This variety of algorithms facilitates partitioning with small tolerance, allowing Capo to distribute the available whitespace uniformly [15] so as to facilitate routing. This provides a convenient baseline for further wirelength improvement [4] by non-uniform distribution (this configuration is now used by default).

The efficiency of the partitioners and placers implemented in Capo as well as the min-cut placement framework are directly responsible for Capo's speed and scalability. To this end, large-scale partitioning is performed in $O(P \log P)$ time, where P is the number of pins in the hypergraph. The overall run-time spent on middle-range partitioning (FM) scales linearly, and so do cumulative run-times of all calls to optimal partitioning and placement. Further complexity analysis shows that Capo's asymptotic run-time scales as $O(P \log^2 P)$ on standard-cell designs.

3 Floorplacement

From an optimization point of view, floorplanning and placement are very similar problems – both seek non-overlapping placements to minimize wirelength. They are mostly distinguished

```
 Variables: queue of placement bins
 Initialize queue with top-level placement bin
1  While (queue not empty)
2    Dequeue a bin
3    If (bin has large/many macros or is marked as merged)
4      Cluster std-cells into soft macros
5      Use fixed-outline floorplanner to pack
         all macros (soft+hard)
6      If fixed-outline floorplanning succeeds
7        Fix macros and remove sites underneath the macros
8      Else
9        Undo one partition decision. Merge bin with sibling
10       Mark new bin as merged and enqueue
11   Else if (bin small enough)
12     Process end case
13   Else
14     Bi-partition the bin into smaller bins
15     Enqueue each child bin
```

Fig. 2. Min-cut floorplacement [34]. Bold-faced lines 3-10 are different from traditional min-cut placement. ©2006 IEEE.

by scale and the need to account for shapes in floorplanning, which calls for different optimization techniques. Netlist partitioning is often used in placement algorithms, where geometric shapes of partitions can be adjusted. This considerably blurs the separation between partitioning, placement and floorplanning, raising the possibility that these three steps can be performed by one CAD tool. The authors of [34] develop such a tool and term the unified layout optimization *floorplacement* following Steve Teig's keynote speech at ISPD 2002.

Min-cut placers scale well in terms of runtime and wirelength minimization, but cannot produce non-overlapping placements of modules with a wide variety of sizes. On the other hand, annealing-based floorplanners can handle vastly different module shapes and sizes, but only for relatively few (100-200) modules at a time. Otherwise, either solutions will be poor or optimization will take too long to be practical. The loose integration of fixed-outline floorplanning and standard-cell placement proposed in [3] suffers from a similar drawback because its single top-level floorplanning step may have to operate on numerous modules. Bottom-up clustering can improve the scalability of annealing, but not sufficiently to make it competitive with other approaches. The work in [34] applies min-cut placement as much as possible and delays explicit floorplanning until it becomes necessary. In particular, since min-cut placement generates a slicing floorplan, it is viewed as an implicit floorplanning step, reserving explicit floorplanning for "local" non-slicing block packing.

Placement starts with a single placement bin representing the entire layout region with all the placeable objects initialized at the center of the bin. Using min-cut partitioning, the bin is split into two bins of similar sizes, and during this process the cut-line is adjusted according to actual partition sizes. Applying this technique recursively to bins (with terminal propagation) produces a series of gradually refined slicing floorplans of the entire layout region. In very small bins, all cells can be placed by a branch-and-bound end-case placer [13]. However, this scheme breaks down on modules that are larger than their bins. When such a module appears

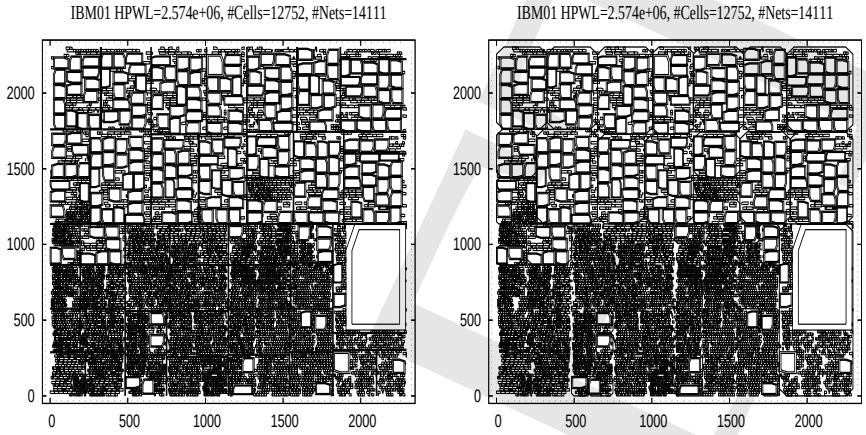


Fig. 3. Progress of mixed-size floorplacement on the IBM01 benchmark from `IBM-MSwPins` [34]. The picture on the left shows how the cut lines are chosen during the first six layers of min-cut bisection. On the right is the same placement but with the floorplanning instances highlighted by "rounded" rectangles. Floorplanning failures can be detected by observing nested rectangles.

in a bin, recursive bisection cannot continue, or else will likely produce a placement with overlapping modules. Indeed, the work in [26] continues bisection and resolves resulting overlaps later. In this technique, one switches from recursive bisection to "local" floorplanning where the fixed outline is determined by the bin. This is done for two main reasons: (i) to preserve wirelength [9], congestion [8] and delay [21] estimates that may have been performed early during top-down placement, and (ii) avoid legalizing a placement with overlapping macros.

While deferring to fixed-outline floorplanning is a natural step, successful fixed-outline floorplanners have appeared only recently [2]. Additionally, the floorplanner may fail to pack all modules within the bin without overlaps. As with any constraint-satisfaction problem, this can be for two reasons: either (i) the instance is unsatisfiable, or (ii) the solver is unable to find any of existing solutions. In this case, the technique undoes the previous partitioning step and merges the failed bin with its sibling bin, whether the sibling has been processed or not, then discards the two bins. The merged bin includes all modules contained in the two smaller bins, and its rectangular outline is the union of the two rectangular outlines. This bin is floorplanned, and in the case of failure can be merged with its sibling again. The overall process is summarized in Figure 2 and an example is depicted in Figure 3.

It is typically easier to satisfy the outline of a merged bin because circuit modules become relatively smaller. However, Simulated Annealing takes longer on larger bins and is less successful in minimizing wirelength. Therefore, it is important to floorplan at just the right time, and the algorithm determines this point by backtracking. Backtracking does incur some overhead in failed floorplan runs, but this overhead is tolerable because merged bins take

considerably longer to floorplan. Furthermore, this overhead can be moderated somewhat by careful prediction.

For a given bin, a floorplanning instance is constructed as follows. All connections between modules in the bin and other modules are propagated to *fixed terminals* at the periphery of the bin. As the bin may contain numerous standard cells, the number of movable objects is reduced by conglomerating standard cells into soft placeable blocks. This is accomplished by a simple bottom-up connectivity-based clustering [25]. The existing large modules in the bin are usually kept out of this clustering. To further simplify floorplanning, soft blocks consisting of standard cells are artificially downsized, as in [4]. The clustered netlist is then passed to the fixed-outline floorplanner Parquet [2], which sizes soft blocks and optimizes block orientations. After suitable locations are found, the locations of all large modules are returned to the top-down placer and are considered fixed. The rows below those modules are fractured and their sites are removed, i.e., the modules are treated as fixed obstacles. At this point, min-cut placement resumes with a bin that has no large modules in it, but has somewhat non-uniform row structure. When min-cut placement is finished, large modules do not overlap by construction, but small cells sometimes overlap (typically below 0.01% by area). Those overlaps are quickly detected and removed with local changes.

Since the floorplacer includes a state-of-the-art floorplanner, it can natively handle pure block-based designs. Unlike most algorithms designed for mixed-size placement, it can pack blocks into a tight outline, optimize block orientations and tune aspect ratios of soft blocks. When the number of blocks is very small, the algorithm applies floorplanning quickly. However, when given a larger design, it may start with partitioning and then call fixed-outline floorplanning for separate bins. As recursive bisection scales well and is more successful at minimizing wirelength than annealing-based floorplanning, the proposed approach is scalable and effective at minimizing wirelength.

Empirical boundary between placement and floorplanning. By identifying the characteristics of placement bins for which the algorithm calls floorplanning, one can tabulate the empirical boundary between placement and floorplanning. Formulating such *ad hoc* thresholds in terms of dimensions of the largest module in the bin, etc., allows one to avoid unnecessary backtracking and decrease the overhead of floorplanning calls that fail to satisfy the fixed outline constraint because they are issued too late. In practice, issuing floorplanning calls too early (i.e., on larger bins) increases final wirelength and sometimes runtime. To improve wirelength, the *ad hoc* tests for large modules in bins (that trigger floorplanning) are deliberately conservative.

These conditions were derived by closely monitoring the legality of floorplanning and min-cut placement solutions. When a partitioned bin yields an illegal placement solution it is clear that the bin should have been floorplanned and a condition should be derived. When a call to floorplanning fails to satisfy the fixed outline constraint the placer has to backtrack. To avoid paying this penalty, a condition should be derived to allow for floorplanning the parent bin and prevent the failure.

These conditions are refined to prevent floorplanning failure by visual inspection of a plot of the resulting parent bin and formulating a condition describing its composition. An example of such a plot is shown in Figure 3. Floorplanned bins are outlined with rounded rectangles. Nested rectangles indicate a failed floorplan run, followed by backtracking and floorplanning of the larger parent bin. In our experience, these tests are strong enough to ensure that at most one level of backtracking is required to prevent overlaps between large modules.

Table 1. Floorplanning conditions used in floorplacement [34]. Test 1 is the most fundamental, for if a bin meeting test 1 were not floorplanned, a failure would be guaranteed at the next level. Tests 2-6 detect bins dominated by large macros. Test 7 is a base case where only one module exists, but it is large.

Floorplanning conditions for floorplacement
N, n: The numbers of large modules and movable objects in a given bin. $A(m)$: The area of the m largest modules in a given bin, $m \leq n$. C: The capacity of a given bin.
Test 1. At least one large module does not fit into a potential child bin.
Test 2. $N \leq 30$ and $A(N) < 0.80 * A(n)$ and $A(n) > 0.6 * C$. Test 3. $N \leq 15$ and $A(N) < 0.95 * A(n)$ and $A(n) > 0.6 * C$.
Test 4. $A(50) < 0.85 * C$. Test 5. $A(10) < 0.60 * C$. Test 6. $A(1) < 0.30 * C$ and $N = 1$.
Test 7. $N = n = 1$.

4 Flexible Whitespace Allocation

The min-cut bisection based placement framework offers much flexibility in whitespace allocation. This section describes uniform allocation of whitespace for min-cut bisection placement and two more sophisticated whitespace allocation techniques, minimum local whitespace and safe whitespace, that can be used for non-uniform whitespace allocation and satisfying whitespace constraints [38].

Uniform Whitespace. A natural scheme for managing whitespace in top-down placement, uniform whitespace allocation, was introduced and analyzed in [15]. Let a placement bin which is going to be partitioned have *site area S*, *cell area C*, *absolute whitespace* $W = \max\{S - C, 0\}$ and *relative whitespace* $w = W/S$. A bi-partitioning divides the bin into two child bins with *site areas* S_0 and S_1 such that $S_0 + S_1 = S$ and *cell areas* C_0 and C_1 such that $C_0 + C_1 = C$. A partitioner is given cell area targets T_0 and T_1 as well as a tolerance τ for a particular bi-partitioning instance. In many cases of bi-partitioning, $T_0 = T_1 = \frac{C}{2}$, but this is not always true [6]. τ defines the maximum percentage by which C_0 and C_1 are allowed to differ from T_0 and T_1, respectively.

The work in [15] bases its whitespace allocation techniques on *whitespace deterioration*: the phenomenon that discreteness in partitioning and placement does not allow for exact uniform whitespace distribution. The whitespace deterioration for a bi-partitioning is the largest α, such that each child bin has at least αw relative whitespace. Assuming non-zero relative whitespace in the placement bin, α should be restricted such that $0 \leq \alpha \leq 1$ [15]. The authors note that $\alpha = 1$ may be overly restrictive in practice because it induces zero tolerance on the partitioning instance but $\alpha = 0$ may not be restrictive enough as it allows for child bins with zero whitespace, which can improve wirelength but impair routability [15].

For a given block, feasible ranges for partition capacities are uniquely determined by α. The partitioning tolerance τ for splitting a block with relative whitespace w is $\frac{(1-\alpha)w}{1-w}$ [15]. The challenge is to determine a proper value for α. First assume that a bin is to be partitioned horizontally n times more during the placement process. n can be calculated as $\lceil \log_2 R \rceil$ where R is the number of rows in the placement bin [15]. Assuming end-case bins have $\alpha = 0$ since they are not further partitioned, $\overline{w}$, the relative whitespace of an end-case bin, is determined to be $\frac{\overline{\tau}}{\overline{\tau}+1}$ where $\overline{\tau}$ is the tolerance of partitioning in the end-case bin [15].

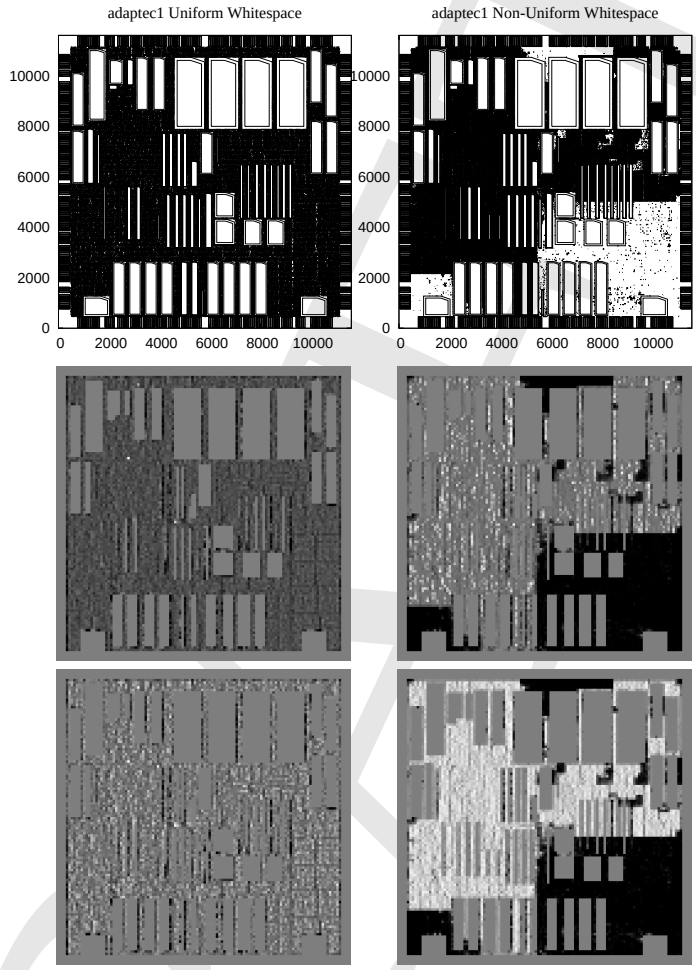


Fig. 4. The top row shows Capo 10 global placements of the contest benchmark adaptec1 with uniform whitespace allocation (left) and non-uniform whitespace allocation (right). Fixed obstacles are drawn with double lines. The middle and bottom rows depict the local utilization the placements. Lighter areas of the placement signify regions that violate the target placement density whereas darker areas have utilization below the target. Areas with no placeable area (such as those with fixed obstacles) are shaded as if they exactly meet the target density. The target placement density for the middle row is 90% and the bottom row is 60% (adaptec1 has 57.34% utilization). The HPWL for the uniform and non-uniform placements are 10.7e7 and 9.0e7 respectively. As the intensity maps show, when 60% utilization is the target, uniform whitespace allocation is much more appropriate than 12% minimum local whitespace. On the other hand, 12% minimum local whitespace has much better wirelength is appropriate when the target is 90% utilization.

Assuming that α remains the same during all partitioning of the given bin gives a simple derivation of $\alpha = \sqrt[n]{\frac{\overline{w}}{w}}$ [15]. A more practical calculation assumes instead that τ remains the same over all partitionings. This leads to $\tau = \sqrt[n]{\frac{1-\overline{w}}{1-w}} - 1$ [15]. $\overline{w}$ can be eliminated from the equation for τ and a closed form for α based only w and n is derived to be $\alpha = \frac{\sqrt[n+1]{1-w}-(1-w)}{w(\sqrt[n+1]{1-w})}$ [15].

If a bin has a user-defined "small" amount of whitespace or less, Capo attempts to divide the cell area approximately in half, within a given tolerance. The appropriate partitioning tolerance is chosen based on whitespace deterioration as calculated above. After a partitionment (i.e., a partitioning solution) is computed, the geometric cutline for the bin is positioned so that each side of the cutline has an equal percentage of whitespace. As tolerance is calculated assuming a fixed cutline, the cutline is shifted to make whitespace more uniform. Such whitespace allocation generally produces routable placements, at the cost of increased wirelength.

Minimum Local Whitespace. If a placement bin has more than a user-defined minimum local whitespace (`minLocalWS`), partitioning will define a tentative cut-line that divides the bin's placement area in half. Partitioning targets an equal division of cell area, but is given more freedom to deviate from its target. Tolerance is computed so that with whitespace deterioration, each descendant bin of the current bin will have at least `minLocalWS` [38].

The assumption that the whitespace deterioration, α, in end-case bins is 0 made in [15] and presented in Section 4 no longer applies, so the calculation of α must change. Since we want all child bins of the current bin to have `minLocalWS` relative whitespace, in particular end case bins must have at least `minLocalWS` and thus we may set $\overline{w} =$ `minLocalWS`, instead of a function of τ. Using the assumption that α remain constant during partitioning, α can be calculated directly as $\alpha = \sqrt[n]{\frac{\overline{w}}{w}}$ [15]. With the more realistic assumption that τ remain constant, τ can be calculated as $\tau = \sqrt[n]{\frac{1-\overline{w}}{1-w}} - 1$ [15]. Knowing τ, α can be computed as $\alpha = (\tau+1) - \frac{\tau}{w}$ [15].

After a partitionment is calculated, the cut-line is shifted to ensure that `minLocalWS` is preserved on both sides of the cut-line. If the minimum local whitespace is chosen to be small, one can produce tightly packed placements which greatly improves wirelength.

Safe Whitespace. The last whitespace allocation mode is designed for bins with "large" quantities of whitespace. In safe whitespace allocation, as with minimum local whitespace allocation, a tentative geometric cut-line of the bin is chosen, and the target of partitioning is an equal bisection of the cell area. The difference in safe whitespace allocation mode is that the partitioning tolerance is much higher. Essentially, any partitioning solution that leaves at least `safeWS` on either side of the cut-line is considered legal. This allows for very tight packing and reduces wirelength, but is not recommended for congestion-driven placement [38].

Figure 4 illustrates uniform and non-uniform whitespace allocation. The top row shows global placements with uniform (left) and non-uniform (right) whitespace allocation on the ISPD 2005 contest benchmark adaptec1 (57.34% utlization) [30]. In the non-uniform placement shown, the minimum local whitespace is 12% and safe whitespace is 14%. The middle and bottome rows show intensity maps of the local utilization of each placement. Lighter areas of the intensity maps signify violations of a given target placement density; darker areas have utilization below the target. Regions completely occupied by fixed obstacles are shaded as if they exactly meet the target density. The target densities for the middle and bottom rows are 90% and 60%, respectively. Note that uniform whitespace produces almost no violations when the target is 90% and relatively few when the target is 60%. The non-uniform placement

has more violations as compared to the uniform placement especially when the target is 60%, but remains largely legal with the 90% target density.

5 Detail Placement

Capo uses several different techniques to further reduce HPWL after global placement such as the sliding window optimizer RowIroning and a greedy cell movement scheme described below. In addition, Capo 10 performs optimal whitespace allocation using min-cost network flows without changing relative cell ordering [7, 39].

RowIroning. In RowIroning, optimal placers based on branch-and-bound and dynamic programming techniques replace windows of cells and whitespace chosen from the placement area [12]. These placers pack cells, and whitespace is represented by fake cells. To model whitespace accurately, one fake cell per site is needed, but Capo evenly divides contiguous regions of whitespace into at most three fake cells to limit runtime. This window of local improvement moves over all cells in left-to-right and top-to-bottom order (or the opposite directions).

Optimal Branch-and-Bound Placement. In the top-down partitioning based placement approach, the original placement problem (considered as a "bin") is partitioned into two subproblems (sub-bins) and then recursively into smaller and smaller subproblems (recall Figure 1). Eventually, wirelength can be directly optimized for bins with few nodes. We now describe *optimal placers* that operate on arbitrary single-row end-case instances given by:[1]

- A hypergraph with nodes (cells) having (x,y)-dimensions. All cell heights are assumed equal to the row height.
- Every hyperedge has a bounding box of fixed pin locations corresponding to the external terminals incident to that net.
- Each hyperedge-to-node connection has a *pin offset* relative to the cell origin.
- A placement region, i.e., a subrow of a certain length.[2]

Additionally assuming the uniform distribution of whitespace, we can consider placement solutions as permutations of hypergraph nodes. The end-case placement problem thus naturally lends itself to enumeration and branch-and-bound. Implementations based on enumeration do not appear competitive in this context and will not be covered further.

In our branch-and-bound placer, nodes are added to the placement one at a time, and the bounding boxes of incident edges are extended to include the new pin locations. The branch-and-bound approach relies on computing, from a given partial placement, a lower bound on the wirelength of any completion of the placement. Extensions of the current partial solution are considered only as long as this lower bound is smaller than the cost of the best seen complete solution.

One difficulty in applying branch-and-bound to end-case placement is varying cell widths. We restrict cells in the small instance to be packed with a fixed-size space between neighbors, i.e., whitespace is distributed equally between them. Replacing a cell with a cell of different width will change the location of at least one neighbor, triggering bounding box recomputations for incident nets. To simplify maintenance, the nodes are packed from left to right and

[1] End-cases have only one row because Capo preferentially splits small multi-row blocks between rows.

[2] For unfortunately short subrows that cannot accommodate all cells without overlaps, our end-case placer first minimizes overlap, then wirelength.

Single Row Placement Branch-and-Bound Input and Data Structures		
Input	cellWidth[0..N]	width of each cell
	pinOffsets[cellId][netId]	pin-offsets for each cell-pin pair
	terminalBoxes[netId]	bounding boxes of net terminals
	RowBox	bounding box of the row
Data Struct	nodeQueue =[0....N-1]	inverse initial ordering
	nodeStack=< *empty* >	placement ordering
	counterArray=< *empty* >	loop counter array
	idx=$N-1$	index
	costSoFar= 0	cost of the current placement
	bestYetSeen = Infi nite	cost of best placement yet found
	nextLoc = row's left edge	location to place next cell at

Single-Row Placement with Branch-and-Bound : Algorithm

```
1  while(idx < numCells)
2  {
3    s.push(q.dequeue()) // add a cell at nextLoc (the right end)
4    c[idx] = idx
5    costSoFar = costSoFar + cost of placing cell s.top()
6    nextLoc.x = nextLoc.x + cellWidth[s.top()]
7
8    if(costSoFar ≤ bestCostSeen) bound
9     c[idx] = 0
10
11   if(c[idx] == 0) // the ordering is complete or has been bounded
12   {
13    if(idx == 0 and costSoFar < bestCostSeen)
14    {
15     bestCostSeen = costSoFar
16     save current placement
17    }
18    while(c[idx] == 0)
19    {
20     costSoFar = costSoFar - cost of placing cell s.top()
21     nextLoc.x = nextLoc.x - cellWidth[s.top()]
22     q.enqueue(s.pop()) // remove the right-most cell
23     idx++
24     c[idx]- -
25    }
26   }
27   idx- -
28 }
```

Fig. 5. Branch-and-Bound algorithm for single-row placement is produced from a lexicographic enumeration of placement orderings by adding code for *bounding* in lines 8 and 9 (in bold) [13]. ©2000 IEEE.

always added to or removed from the right end of the partially-specified permutation. Such a lexicographic ordering naturally leads to a stack-driven implementation, where the states of incident nets are "pushed" onto stacks when a node is appended on the right side of the ordering, and "popped" when the node is removed. Bounding entails "popping" nodes at the end of a partial solution before all lexicographically greater partial solutions have been visited. Pseudocode is provided in Figure 5.

Greedy Cell Movement. Capo makes use of a gridded greedy movement technique to improve both wirelength and whitespace distribution. A grid is imposed on the placement region to analyze local placement density. For cells that are in regions with density violations, candidate legal new locations are found in areas of lower density violation. Candidate moves are ranked by how well they alleviate the violations and how they affect wirelength. Moves

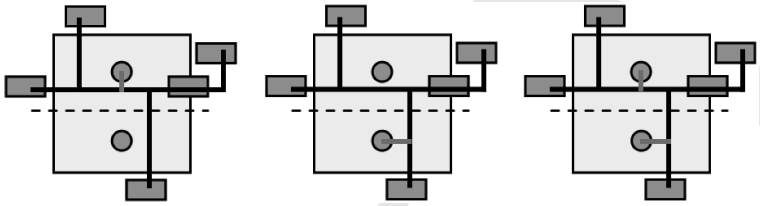


Fig. 6. Calculating the three costs for weighted terminal propagation with StWL: w_1 (left), w_2 (middle), and w_{12} (right) [35]. The net has five fixed terminals: four above and one below the proposed cut-line. For the traditional HPWL objective, this net would be considered inessential. Note that the structure of the three Steiner trees may be entirely different, which is why w_1, w_2 and w_{12} are evaluated independently.

are made until a threshold of improvement is reached. We have found this to be a fast and effective method of removing density violations without adversely affecting wirelength.

6 Placement for Routability

With uniform whitespace allocation, Capo typically produces routable placements, but some congested areas remain. Capo 10 implements a whitespace allocation scheme described in [35] to improve placement routability. This technique uses a congestion map to estimate routing congestion after each layer of min-cut placement. Based on the congestion estimates, whitespace is allocated preferentially to areas of high congestion through cutline shifting. Coupled with other techniques from ROOSTER [35], Capo 10 outperforms best published routed wirelengths and via counts as of Fall 2006.

6.1 Optimizing Steiner Wirelength

Weighted terminal propagation as described in [17] is sufficiently general to account for objectives other than HPWL such as Steiner Wirelength (StWL) [35]. StWL is known to correlate with final routed wirelength (rWL) more accurately than HPWL and the authors of [35] hypothesize that if StWL could be directly optimized during global placement, one may be able to enhance routability and reduce routed wirelength.

When bipartitioning a bin, the pins for a particular net may all fall into one partition (leaving the net uncut) or be split amongst both partitions (cutting the net). We will refer to the two possible partitions as partition 1 and partition 2. When using weighted terminal propagation from [17], one must calculate three costs per net per partitioning instance: w_1, w_2 and w_{12}. These costs represent the cost of the pins of a net all being placed in partition 1, partition 2 or split between both, respectively.

The points required to calculate w_1 for a given net are the terminals on the net (pins not allowed to move) plus the center of partition 1. Similarly, the points required to calculate w_2 are the terminals plus the center of partition 2. Lastly, the points to calculate w_{12} are the terminals on the net plus the centers of both partitions. See Figure 6 for an example of calculating these three costs. Clearly the HPWL of the set of points necessary to calculate w_{12} is at least as large as that of w_1 and w_2 since it contains an additional point. By the same logic, StWL also satisfies this relationship since RSMT length can only increase with additional

points. Since StWL is a valid cost function for these weighted partitioning problems, this is a framework whereby it can be minimized [35].

The simplicity of this framework for minimizing StWL is deceiving. In particular, the propagation of terminal locations to the current placement bin and the removal of inessential nets [13] — standard techniques for HPWL minimization — cannot be used when minimizing StWL. Moving terminal locations drastically changes Steiner-tree construction and can make StWL estimates extremely inaccurate. Nets that are considered inessential in HPWL minimization (where the x- or y-span of terminals, if the cut is vertical or horizontal respectively, contains the x- or y-span of the centers of child bins) are not necessarily inessential when considering StWL because there are many Steiner trees of different lengths that have the same bounding box. Figure 6 illustrates a net that is inessential for HPWL minimization but essential for StWL minimization. Not only computing Steiner trees, but even traversing all relevant nets to collect all relevant point locations can be very time-consuming. Therefore, the main challenge in supporting StWL minimization is to develop efficient data structures and limit additional runtime during placement [35].

Pointsets with multiplicities. Building Steiner trees for each net during partitioning is a computationally expensive task. To keep runtime reasonable when building Steiner trees for partitioning, the authors of [35] introduce a simple yet highly effective data structure — *pointsets with multiplicities*. For each net in the hypergraph, two lists are maintained. The first list contains all the unique pin locations on the net that are fixed. A fixed pin can come from sources such as terminals or fixed objects in the core area. The second list contains all the unique pin locations on the net that are movable, i.e., all other pins that are not on the fixed list. All points on each list are unique so that redundant points are not given to Steiner evaluators which may increase their runtime. To do so efficiently, the lists are kept in a sorted order. For both lists, in addition to the location of the pin, the number of pins that correspond to a given point is also saved [35].

Maintaining the number of actual pins that correspond to a point in a pointset (the multiplicity of that point) is necessary for efficient update of pin locations during placement. If a pin changes position during placement, the pointsets for the net connected to the pin must be updated. First, the original position of the pin must be removed from the movable point set. As multiple pins can have the same position, especially early in placement, the entire net would need to be traversed to see if any other pins share the same position as the pin that is moving. Multiplicities allow to know this information in constant time. To remove the pin, one performs a binary search on the pointset and decreases the multiplicity of the pin's position by 1. If this results in the position having a multiplicity of 0, the position can be removed entirely. Insertion of the pin's new position is similar: first, a binary search is performed on the pointset. If the pin's position is already present in the pointset, the multiplicity is increased by 1. Otherwise, the position is added in sorted order with a multiplicity of 1. Empirically, building and maintaining the pointset data structures takes less than 1% of the runtime of global placement [35].

Performance. We compared three Steiner evaluators in terms of runtime impact and solution quality. They chose the FastSteiner [22] evaluator for global placement based on its reasonable runtime and consistent performance on large nets. Empirical results show the use of FastSteiner leads to a reduction of StWL by 3% on average on the IBMv2 benchmarks [41] (with a reduction of routed wirelength up to 7%) while using less than 30% additional runtime [35].

6.2 Congestion-based Cutline Shifting

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

After the partitioner returns a solution, we have the flexibility to keep the cutline as it was chosen before partitioning or to change it to optimize an objective. The WSA [27] technique, applied after placement, geometrically divides the placement area in half and estimates the congestion in both halves of the layout. It then allocates more area to the side with greater routing demand, i.e. shifts the cutline, and proceeds recursively on the two halves of the design. In WSA, cells must be re-placed after the whitespace allocation. However, we can avoid this re-placement because our cells have not yet been placed and will be taken care of naturally during the min-cut process.

Cutline shifting used to handle congestion necessitates a slicing floorplan. The only work in the literature that describes top-down congestion estimates and uses them in placement assumes a grid structure [8]. Therefore we develop the following technique: before each round of partitioning, we overlay the entire placement region on a grid. We choose the grid such that each placement bin is covered by 2-4 grid cells. We then build a congestion map using the last updated locations of all pins. We choose the mapping technique from [40] as it shows good correlation with routed congestion.

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

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

7 Improved RTL Placement

Industrial floorplacement problems are increasingly difficult due to factors such as an increasing number of movable modules and a wide variation of module sizes. There is also insufficient cohesion for whitespace allocation between top-down methods and macro-placement algorithms. For example, a partitioner may misapproximate the area required by a set of macros and incorrectly allocate whitespace. To address these issues, we have integrated into Capo 10 the SCAMPI (SCalable Advanced Macro Placement Improvements) work [31]. The top-down partitioning flow is modified to selectively place large macros, while smaller macros are clustered into soft modules that will be placed later. The robustness of the flow is also improved by employing fast *look-ahead* Simulated Annealing on large macros of newly created bins.

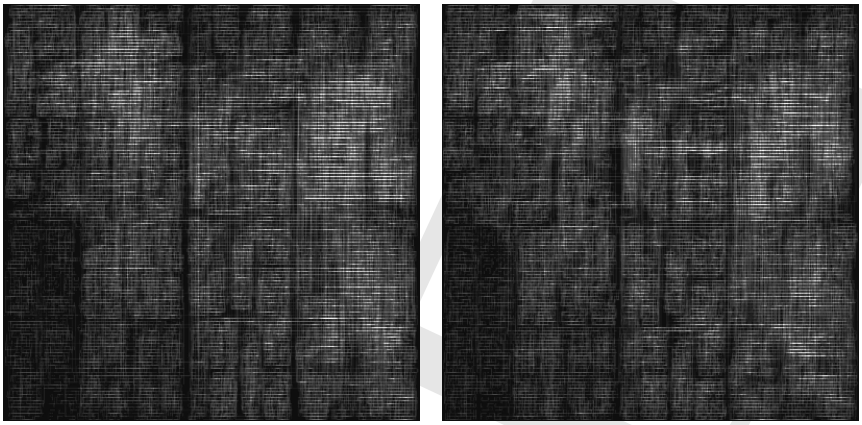


Fig. 7. Congestion maps for the ibm01h benchmark: uniform whitespace allocation (produced with Capo -uniformWS) is illustrated on the left, congestion-driven allocation in ROOSTER is illustrated on the right [35]. The peak congestion when using uniform whitespace is 50% greater than that for our technique. When routed with Cadence WarpRoute, uniform whitespace produces 3.95% overfull global routing cells and routes in just over 5 hours with 120 violations. ROOSTER's whitespace allocation produces 3.18% overfull global routing cells and routes in 22 minutes without violations.

This allows early detection of bins difficult to floorplan, and alerts the placer to backtrack and seek a different partitioning solution.

Selective fborplanning for multi-million gate designs. One case that is not considered by either the original floorplacement techniques [34] or those introduced in SCAMPI [31] is where there are an extreme number of movable modules and an extreme ratio between the largest and smallest macro. An example of this is the *newblue1* benchmark from the ISPD'06 placement contest suite. The *newblue1* benchmark contains 64 macros and 330073 standard cells. As we show below, such a configuration is problematic for floorplacement tools.

Recall that a floorplacer utilizes a floorplanner to place macros. As the floorplanner uses Simulated Annealing to pack blocks, clustering is performed on the netlist to improve scalability. However, a very large number of small modules may stress clustering algorithms, which, in the absence of refinement, may undermine the overall solution quality.[3]

Figure 9 shows the *newblue1* benchmark placed with SCAMPI, before and after our most recent improvements. In the original SCAMPI flow, the large block was designated for floorplanning by Parquet at the top level. Parquet precedes annealing with clustering to reduce the size of the netlist. However, given the large number of small modules, the simple-minded clustering algorithm in Parquet ended up taking 16% of total runtime, whereas annealing took only 4%. Additionally, even if clustering were more scalable, clustering such a large number of small macros into large, soft macros can lead to unnatural or unrepresentative netlists. In the original SCAMPI flow, the clusters formed by the standard cells in *newblue1* became large

[3] Refinement algorithms would need to operate on very large netlists and may require long runtimes.

```
 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 fborplanning on partition
5      If look-ahead fborplanning 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 fborplanning
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 fborplanning 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
```

Fig. 8. Our modified min-cut floorplacement flow [31]. Bold-faced lines are new compared to [34].

enough to artificially constrain the movement of the large macro during floorplanning. This is mainly a limitation of Simulated Annealing as it becomes impractical in solution quality and runtime for over 100 modules.

Therefore, we propose the following method. Whenever a bin is designated for floorplanning and the largest real module is smaller in area than the largest soft macro built from clustering (this area can be estimated without actually performing clustering), we do not use Simulated Annealing. Instead, a simple analytical placement technique, such as *Successive Over-Relaxation* (SOR), is used to determine reasonable locations for the large macros.[4] It has been shown that analytical techniques are good at finding general areas where objects should be placed [6], so this is a reasonable and efficient solution for placing a large macro or macros in this situation. As such, this technique may also be useful in regions with large amounts of whitespace as block-packing often overlooks good solutions in such situations. Objectives other than HPWL, such as routing congestion and timing, are also important, and any analytical placer used in this context should place macros with respect to the most relevant objective(s). Our key observation is that placing such macros early is helpful.

[4] Any analytical placement technique can be used, but SOR may be sufficient since we are not necessarily looking for a non-overlapping placement. For example, we have also used a linearized version of the SOR technique as well and seen improvements in HPWL at the expense of moderately increased runtime.

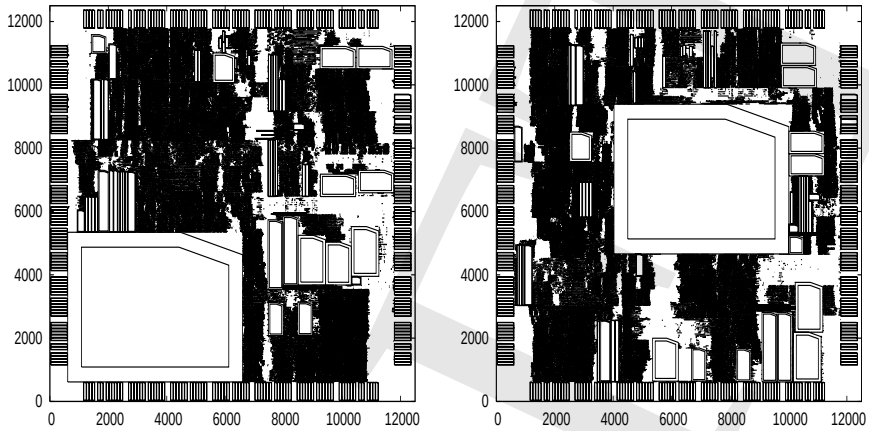


Fig. 9. The *newblue1* benchmark placed by SCAMPI before (left) and after (right) our recent modifications. Before our improvements to SCAMPI, the clusters formed by the smaller modules at the top level constrain the movement of the largest module and result in it being placed in the bottom-left corner of the core. After our improvements, the largest macro is placed using Successive Over-Relaxation (SOR).

When there is only one large macro to be placed, the solution of the analytical tool is used and the macro is fixed in its desired location. To place a small number of large macros with this method, we again compute macro locations with the analytical tool, but must legalize the macro locations to maintain the correct-by-construction paradigm of floorplacement. Overlaps can be legalized in several ways. One way is to use a greedy macro legalization technique such as the macro legalizer described in [34, Section 3.3]. Another method for removing macro overlap is the constraint-based floorplan repair algorithm FLOORIST [29]. Following legalization, one can shift the macros so that their center of mass coincides with their center of mass before legalization in keeping with the spirit of the analytical placement. This technique contributed to HPWL improvement over the ISPD 2006 Placement Contest results of Capo by 17% on newblue1, with an overall improvement in the contest score on the ISPD 2006 benchmark suite by 10%, moving Capo three positions higher.

Temporary Macro Deflation. Low-whitespace conditions in block-packing instances formed during floorplacement can worsen solution quality significantly. In such cases, the block-packing engine focuses mainly on finding legal solutions rather than those that have good wirelength. In addition, a legal solution may not be found which leads to backtracking and increased runtime as well. To improve the solution quality of block-packing instances created during floorplacement, we prevent these low-whitespace conditions.

To account for standard cells in the floorplacement framework, standard cells are clustered into soft blocks for instances of block-packing [1]. To improve the likelihood of finding a legal fixed-outline solution, these soft blocks representing standard cells are reduced in size [1]. We propose extending this deflation to include hard blocks in addition to soft blocks. When a block-packing instance is formed, we adjust the sizes of hard blocks to maintain a minimum amount of whitespace. All blocks in the instance are sized in the same way and aspect ratios

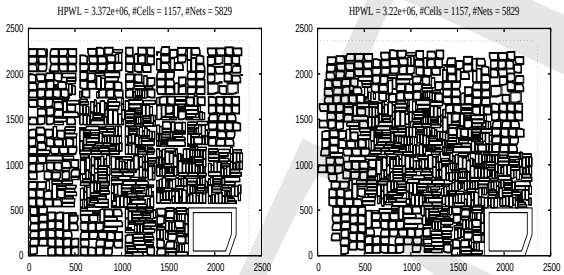


Fig. 10. A placement of the ibm-HB01 benchmark produced by Capo 9.4 that exhibits an overly generous whitespace allocation scheme in Capo. After re-allocating whitespace with a min-cost max-flow technique, we decrease HPWL by 4.5%.

are maintained. The resized instance, made easier by the addition of whitespace, is placed using Simulated Annealing as normal.

Resizing the hard blocks in this way has the positive effect of making fixed-outline block-packing easier, which allows the block-packing engine to focus on HPWL minimization rather than mere legality in cases where whitespace is limited, but removes the correct-by-construction property upon which floorplacement is built. To alleviate this problem, we apply legalization to macros after packing. We use the fast and robust constraint-based floorplan repair algorithm FLOORIST [29] after each layer of placement where block-packing took place. FLOORIST moves macros minimally when repairing macro overlaps, so the reduced HPWL found in easier block-packing instances is preserved.

Empirically we find that the overhead of running FLOORIST for legalization is mitigated by the fact that block-packing is easier and therefore faster. In terms of solution quality, we find that temporary macro deflation reduces HPWL by 2-3%.

Whitespace Re-allocation Using Linear Programming and Min-cost Max-flow. As we have noted earlier, in order to avoid cases of backtracking which can dramatically increase both HPWL and runtime, Capo allocates whitespace uniformly during partitioning when macros are present. We have shown in Figure 10 this whitespace allocation scheme can lead to HPWL that is much larger than a tighter packing. In order to reclaim some of the HPWL lost due to uniform distribution during global placement, we propose a technique to re-allocate whitespace during detail placement.

Our technique builds upon the well-known linear programming formulations used, e.g., in [39] and [33] in that we impose linear constraints for movable objects based on their relative positions with respect to core boundaries and other movable objects. More details on the linear programming formulation such as types of constraints and the objective function are given below. We include additional linear inequalities to account for fixed obstacles and region constraints. One major difference from previous work is that we guarantee that the x and y locations found align to legal sites and rows, as explained later in this section.

We handle re-allocation of whitespace separately for the horizontal and vertical directions, and preserve local relative ordering of movables in each direction. In other words, movable objects may not jump over each other or any fixed obstacles when whitespace is being re-allocated. Unlike in global-placement [33], we start with legal or nearly-legal locations. This simplifies our selection of relative constraints to include into the LP formulation as follows. In the horizontal case, we examine each row individually. For each cell or macro that intersects

the row, we determine its immediate neighbors to the left and to the right (those objects with which the current object could feasibly overlap if it would slide to the left or right). These neighbors can include movable objects, row or region boundaries as well as fixed obstacles. After the neighborhood relations are determined, we constrain an object to lie between its left- and right-hand neighbors. Construction of constraints for the vertical case is analogous where rows are replaced with columns and site width is replaced by row height. Unlike the formulation from [33], ours guarantees an overlap-free placement and needs to be solved only once. In contrast with [39], we include only several constraints per movable object rather than a quadratic number of constraints read from a sequence-pair. This significantly improves scalability and allows one to pack more tightly.

In addition to the constraints above, we minimize HPWL. This is done by adding $x_{min}, x_{max}, y_{min}, y_{max}$ variables for each net, and the terms $(x_{max} - x_{min})$ and $(y_{max} - y_{min})$ to the objective function. To solve the entire LP efficiently, we dualize it as in [39] and cast the dual as a min-cost max-flow instance. The latter is solved using the scaling push-relabeling algorithm of Goldberg [19]. An important feature of our technique is the use of integrality of the solutions found by this algorithm — we scale the coordinates so that integer x values correspond to legal sites and integer y values correspond to standard-cell rows. Figure 10 illustrates whitespace re-allocation in the horizontal and vertical directions applied to a placement of the ibm-HB01 benchmark. HPWL is improved by 4.5% while runtime of the technique is less than 1% of placement runtime.

8 Incremental Placement

To develop a strong incremental placement tool, ECO-system, we build upon an existing global placement framework and must choose between analytical and top-down. The main considerations include robustness, the handling of movable macros and fixed obstacles, as well as consistent routability of placements and the handling of density constraints. Based on recent empirical evidence [31, 35, 38], the top-down framework appears a somewhat better choice. Indeed the 2 out of 9 contestants in the ISPD 2006 Competition that satisfied density constraints were top-down placers. However, analytical algorithms can also be integrated into our ECO-system when particularly extensive changes are required. ECO-system favorably compares to recent detail placers in runtime and solution quality and fares well in high-level and physical synthesis.

General Framework. The goal of ECO-system is to reconstruct the internal state of a min-cut placer that could have produced a given placement **without the expense of global placement**. Given this state, we can choose to accept or reject previous decisions based on our own criteria and build a new placement for the design. If many of the decisions of the placer were good, we can achieve a considerable runtime savings. If many of the decisions are determined to be bad, we can do no worse in terms of solution quality than placement from scratch. An overview of the application of ECO-system to an illegal placement is depicted in Figure 12. See the algorithm in Figure 11.

To rebuild the state of a min-cut placer, we must reconstruct a series of cut-lines and partitioning solutions efficiently. To extract a cut-line and partitioning solution from a given placement bin, we examine all possible cut-lines as well as the partitions they induce. We start at one edge of the placement bin (left edge for a vertical cut and bottom edge for a horizontal cut) and move towards the opposite edge. For each potential cut-line encountered, we maintain the cell area on either side of the cut-line, the partition induced by the cut-line and the net cut.

```
  Variables: queue of placement bins
  Initialize queue with top-level placement bin
1  While(queue not empty)
2    Dequeue a bin
3    If(bin not marked to place from scratch)
4     If(bin overfull)
5      Mark bin to place from scratch, break
6     Quickly choose the cut-line which has
        the smallest net-cut considering
        cell area balance constraints
7     If(cut-line causes overfull child bin)
8      Mark bin to place from scratch, break
9     Induce partitioning of bin's cells from cut-line
10    Improve net-cut of partitioning with
        single pass of Fiduccia-Mattheyses
11    If(% of improvement > threshold)
12     Mark bin to place from scratch, break
13    Create child bins using cut-line and partitioning
14    Enqueue each child bin
15   If(bin marked to place from scratch)
16    If(bin small enough)
17      Process end case
18    Else
19      Bi-partition the bin into child bins
20       Mark child bins to place from scratch
21      Enqueue each child bin
```

Fig. 11. Incremental min-cut placement [36]. Bold-faced lines 3-15 and 20 are different from traditional min-cut placement. ©2007 IEEE.

Fast Cut-line Selection. For simplicity, assume that we are making a vertical cut and are moving the cut-line from the left to the right edge of the placement bin (the techniques necessary for a horizontal cut are analogous). Pseudo-code for choosing the cut-line is shown in Figure 13. To find the net cut for each possible cut-line efficiently, we first calculate the bounding box of each net contained in the placement bin from the original placement. We create two lists with the left and right x-coordinates of the bounding boxes of the nets and sort them in increasing x-order. While sliding the cut-line from left to right (in the direction of increasing x-coordinates), we incrementally update the net-cut and amortize the amount of time used to a constant number of operations per net over the entire bin. We do the same with the centers of the cells in the bin to incrementally update the cell areas on either side of the cut-line as well as the induced partitioning. While processing each cut-line, we save the cut-line with smallest cut that is legal given partitioning tolerances. An example of finding the cut-line for a partitioning bin is shown in Figure 12.

Once a partitioning has been chosen, we accept or reject it based on how much it can be improved by **a single pass of a Fiduccia-Mattheyses partitioner with early termination** (which takes only several seconds even on the largest ISPD'05 circuit). [5] The intuition is that if the constructed partitioning is not worthy of reuse, a single Fiduccia-Mattheyses pass could improve its cut non-trivially. If the Fiduccia-Mattheyses pass improves the cut beyond a certain threshold, we discard the solution and bisect the entire bin from scratch. If this test passes, we check legality: if a child bin is overfull, we discard the cut-line and bisect from scratch.

Scalability. Pseudo-code for the cut-line location process used by ECO-system is shown in Figure 13. The runtime of the algorithm is linear in the number of pins incident to the bin,

[5] We do not assume that the initial placement was produced by a min-cut algorithm.

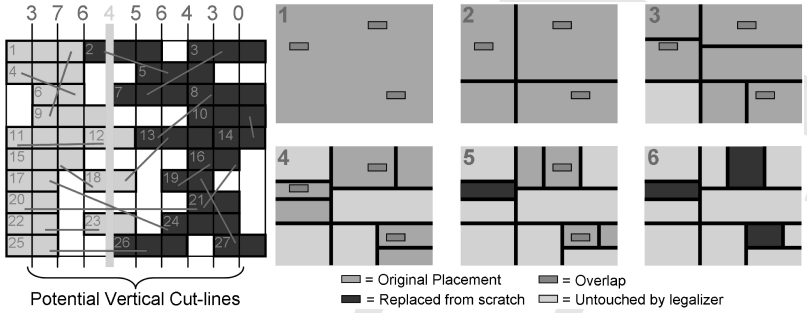


Fig. 12. Fast legalization by ECO-system [36]. The image on the left illustrates choosing a vertical cut-line from an existing placement. Nets are illustrated as red lines. Cells are individually numbered and take 2 or 3 sites each. Cut-lines are evaluated by a left-to-right sweep (net cuts are shown above each line). A cut-line that satisfies partitioning tolerances and minimizes cut is found (thick green line). Cells are assigned to "left" and "right" according to the center locations. On the right, placement bins are subdivided using derived cut-lines until i) a bin contains no overlap and is ignored for the remainder of the legalization process or, ii) the placement in the bin is considered too poor to be kept and is replaced from scratch using min-cut or analytical techniques. ©2007 IEEE.

cells incident contained in the bin, and possible cut-lines for the bin. Since a single Fiduccia-Mattheyses pass takes also takes linear time [18], the asymptotic complexity of our algorithm is linear. If we let P represent the number of pins incident to the bin, C represent the number of cells in the bin and L represent the number of potential cut-lines in the bin, the cut-line selection process runs in $O(P+C+L)$ time. In the vast majority of cases, $P > C$ and $P > L$, so the runtime estimate simplifies to $O(P)$.

The number of bins may double at each hierarchy layer, until bins are small enough for end-case placement. End-case placement is generally a constant amount of runtime for each bin, so it does not affect asymptotic calculations. Assume that ECO-system is able to reuse all of the original placement. Since ECO-system performs bisection, it will have $O(\log C)$ layers of bisection before end-case placement. At layer i, there will be $O(2^i)$ bins, each taking $O\left(\frac{P}{2^i}\right)$ time. This gives a total time per layer of $O(P)$. Combining all layers gives $O(P \log C)$. Empirically, the runtime of the cut-line selection procedure (which includes a single pass of a Fiduccia-Mattheyses partitioner) is much smaller than partitioning from scratch. On large benchmarks, cut-line selection requires 5% of ECO-system runtime time whereas min-cut partitioning generally requires 50% or more of ECO-system runtime.

Handling Macros and Obstacles. With the addition of macros, the flow of top-down placement becomes more complex. We adopt the technique of "floorplacement" which proceeds as traditional placement until a bin satisfies criteria for block-packing [31,34]. If the criteria suggest that the bin should be packed rather than partitioned, a fixed-outline floorplanning instance is induced from the bin where macros are treated as hard blocks and standard cells are clustered into soft blocks. The floorplanning instance is given to a Simulated Annealing-based floorplanner to be solved. If macros are placed legally and without overlap, they are considered fixed. Otherwise, the placement bin is merged with its sibling bin in the top-down

```
Input: placement bin, balance constraint
Output: x-coord of best cut-line
1  numCutlines =
    1+⌊(rightBinEdgeX−leftBinEdgeX)/cellSpacing⌋
2  Create three arrays of size numCutlines:
    LEFT, RIGHT, AREA
3  Set all elements of LEFT, RIGHT, and AREA to 0
4  Foreach net
5    Calculate x-coord of left- and right-most pins
6    leftCutlineIndex =
      max(0,⌈(leftPinX−leftBinEdgeX)/cellSpacing⌉)
7    rightCutlineIndex =
      max(0,⌈(rightPinX−leftBinEdgeX)/cellSpacing⌉)
8    if(leftCutlineIndex < numCutlines)
9      LEFT[leftCutlineIndex]+=1
10   if(rightCutlineIndex < numCutlines)
11     RIGHT[rightCutlineIndex]+=1
12 Foreach cell
13   Calculate x-coord of the center of the cell
14   cutlineIndex =
      max(0,⌈(centerX−leftBinEdgeX)/cellSpacing⌉)
15   if(cutlineIndex < numCutlines)
16     AREA[cutlineIndex]+=cellArea
17 Set X = leftBinEdge, CURCUT = 0, BESTCUT = ∞
   BESTX = ∞, LEFTPARTAREA = 0
18 For(I = 0;I < numCutlines;I+=1,X+=cellSpacing)
19   CURCUT+=LEFT[I]
20   CURCUT−=RIGHT[I]
21   LEFTPARTAREA+=AREA[I]
22   If(CURCUT < BESTCUT and
        LEFTPARTAREA satisfies balance constraint)
23     BESTCUT = CURCUT
24     BESTX = X
25 Return BESTX
```

Fig. 13. Algorithm for finding the best vertical cut-line from a placement bin. Finding the best horizontal cut-line is largely the same process. Note that the runtime of the algorithm is linear in the number of pins incident to the bin, cells incident contained in the bin, and possible cut-lines for the bin.

hierarchy and the merged bin is floorplanned. Merging and re-floorplanning continues until the solution is legal.

We add a new floorplanning criterion for our legalization technique. If no macros in a placement bin overlap each other, we generate a placement solution for the macros of the bin to be exactly their placements in the initial solution. If some of the macros overlap with each other, we let other criteria for floorplanning decide. If block-packing is invoked, we must discard the placement of all cells and macros in the bin and proceed as described in [34].

During the cut-line selection process, some cut-line locations are considered invalid — namely those that are too close to obstacle boundaries but do not cross the obstacles. This is done to prevent long and narrow slivers of space between cut-lines and obstacle boundaries. Ties for cut-lines are broken based on the number of macros they intersect. This helps to reduce overfullness in child bins allowing deeper partitioning, which reduces runtime.

Relaxing Overfullness Constraints. One of the primary objectives of ECO-system is to reuse as much relevant placement information as possible from a given placement. As described above, it is possible to find a cut-line which has a good cut but is not legal due to space

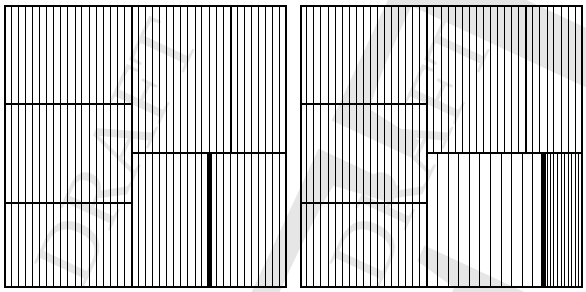


Fig. 14. Shifting a cut-line chosen during ECO cut-line selection. Unlike the WSA technique [27, 28], cut-line shifting during ECO is not done on geometric cut-lines but instead on those cut-lines which are chosen during fast cut-line selection. The image on the left shows a placement that has been divided into bins during the course of ECO-system. In the image on the right, the chosen cut-line of the bottom-right bin is shifted to the right. The density of vertical lines represents the initial placement and its scaling around the moving cutline (shown in red).

constraints. In these cases, ECO-system must discard these good solutions and partition from scratch.

In order to make better use of the given placement, we propose the following addition to ECO-system. In these situations, we allow ECO-system to shift the cut-line to legalize the derived partition with respect to area. Cut-line shifting is a technique commonly used in the top-down min-cut placement for allocation of whitespace [4, 27, 28, 35, 38]. The cut-line is shifted as little as possible to make the derived partitioning legal with respect to area. If it is impossible to find an area-legal cutline, the derived partitioning must be discarded and ECO-system proceeds normally.

If cut-line shifting is successful in correcting the illegality, the original placement must be modified for purposes of consistency. To do so, cells are scaled proportionately within the placement bin based on their original positions, the position of the originally chosen cut-line and the position of the shifted cut-line in a manner similar to that in the WSA technique [27,28]. As the centers of cells are used to determine in what partitions cells belong during fast cut-line selection, we shift cell locations based on center locations as well to ensure that cut-line shifting will not change derived partitions. We seek to shift cell locations and maintain the following property: the relative position between cells before and after shifting is maintained. Also, if a cell were in the middle of a partition before shifting, it should remain in the middle of a partition after shifting. Let x_L and x_R represent the x-coordinates of the left and right sides of the placement bin, x_{orig}^{cut} and x_{new}^{cut} the x-coordinates of the original and new cuts, and, lastly, x_{orig}^{cell} and x_{new}^{cell} the x-coordinates of the center of a particular cell before and after shifting. We wish to maintain the following ratios (for vertical partitioning):

$$\frac{x_{orig}^{cell} - x_L}{x_{orig}^{cut} - x_L} = \frac{x_{new}^{cell} - x_L}{x_{new}^{cut} - x_L},\ x_{orig}^{cell} \leq x_{orig}^{cut}$$

$$\frac{x_R - x_{orig}^{cell}}{x_R - x_{orig}^{cut}} = \frac{x_R - x_{new}^{cell}}{x_R - x_{new}^{cut}},\ x_{orig}^{cell} > x_{orig}^{cut}$$

Solving for x_{new}^{cell} :

$$x_{new}^{cell} = \begin{cases} x_L + \left(x_{orig}^{cell} - x_L\right) \frac{x_{new}^{cut} - x_L}{x_{orig}^{cut} - x_L}, & x_{orig}^{cell} \leq x_{orig}^{cut} \\ x_R - \left(x_R - x_{orig}^{cell}\right) \frac{x_R - x_{new}^{cut}}{x_R - x_{orig}^{cut}}, & x_{orig}^{cell} > x_{orig}^{cut} \end{cases}$$

The new y-coordinates of cells shifted during horizontal partitioning are calculated analogously.

Figure 14 illustrates the scaling involved when a cut-line is shifted. In the figure, the cut-line of the bottom-right bin is shifted to the right. All objects to the left and right of the cut-line are scaled appropriately. Objects that were to the left of the original cut-line remain to the left and are spread out and objects on the right are packed closer together.

Shifting proportionately in this way maintains the relative ordering of all the cells within the current placement bin. Also the partitioning induced by the cutline remains unchanged so ECO-system can proceed as normal. Shifting the cut-line in this manner can allow deeper ECO partitioning which can reduce both runtime and cell displacement.

Satisfying Density Constraints. A common method for increasing the routability of a design is to inject whitespace into regions that are congested [4, 27]. One can also require a minimum amount of whitespace (equivalent to a maximum cell density) in local regions of the design to achieve a similar effect [38]. As one of ECO-system's legality checks is essentially a density constraint (checking to see if a child bin has more cell area assigned to it than it can physically fit), this legality check is easy to generalize. The new criterion for switching from using the initial placement and partitioning from scratch is based on a child bin having less than a threshold percent of relative whitespace, which is controlled by the user.

The cut-line shifting feature of ECO-system can also be used to satisfy density constraints. As ECO-system proceeds, cut-lines can be shifted as described above to implement a variety of whitespace allocation schemes [27, 28, 35, 38]. Specifically, ECO-system can implement the hierarchical whitespace injection of WSA [27, 28]. WSA chooses cut-lines based only on the geometry of a placement bin and shifts these cut-lines from the top down. ECO-system chooses cut-lines that are more natural to the original placement, shifts cut-lines top-down, and also supports fixed objects and movable macros.

9 Memory Profile

Capo's non-uniform whitespace allocation techniques tend to produce unbalanced partitionments at the top layers. As peak memory usage grows with partitioning problem size, memory consumption can stay near the peak for longer periods of time during placement. To counteract the increased possibility of thrashing, Capo 10 has several memory improvements which include the slimming down of data structures and carefully choosing the lifetimes of major data structures so that fewer need to be in main memory simultaneously. The most radical of these changes involves removing the netlist hypergraph from main memory during the largest partitioning instances and rebuilding it from scratch afterwards. These changes reduce peak memory consumption by 2x compared to Capo 9.1 but slow down global placement by 10%.

10 Performance on Publicly-Available Benchmarks

To illustrate Capo's ability to handle a wide range of placement instances, we evaluate Capo on benchmarks with routing information, mixed-size benchmarks and the extremely large benchmarks with generous amounts of whitespace from the ISPD 2005 and 2006 placement competitions.

Table 2. A comparison of ROOSTER to the most recent version of mPL-R + WSA and APlace 2.04 on the IBMv2 benchmarks [41]. All routed wirelengths (rWL) are in meters. "Time" represents routing runtime in minutes. Note that while APlace 2.04 achieves overall smaller wirelength than ROOSTER, it routes with violations on 2 of the 16 benchmarks. Best legal rWL and via counts are in bold.

	ROOSTER				Latest mPL-R + WSA				APlace 2.04 -R 0.5			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
ibm01e	0.733	**122286**	0	42	**0.718**	123064	0	11	0.790	158646	85	132
ibm01h	0.746	**124307**	0	32	**0.691**	213162	0	11	0.732	161717	2	121
ibm02e	2.059	259188	0	13	**1.821**	**250527**	0	11	1.846	254713	0	9
ibm02h	2.004	262900	0	14	**1.897**	**260455**	0	13	1.973	268259	0	14
ibm07e	4.075	**476814**	0	17	4.130	492947	0	21	**3.975**	500574	0	17
ibm07h	4.329	**489603**	0	19	4.240	516929	0	26	**4.141**	518089	0	23
ibm08e	4.242	**559636**	0	17	4.372	579926	0	23	**3.956**	588331	0	18
ibm08h	4.262	**574593**	0	20	4.280	599467	0	26	**3.960**	595528	0	18
ibm09e	3.165	**466283**	0	11	3.319	488697	0	17	**3.095**	502455	0	11
ibm09h	3.187	**475791**	0	11	3.454	502742	0	19	**3.102**	512764	0	12
ibm10e	6.412	**749731**	0	22	6.553	777389	0	30	**6.178**	782942	0	23
ibm10h	6.602	**775018**	0	27	6.474	799544	0	33	**6.169**	801605	0	28
ibm11e	**4.698**	**605807**	0	15	4.917	633640	0	22	4.755	648044	0	18
ibm11h	**4.697**	**618173**	0	16	4.912	660985	0	25	4.818	677455	0	24
ibm12e	9.289	**918363**	0	36	10.185	995921	0	57	**8.599**	921454	0	32
ibm12h	9.289	**938971**	0	43	9.724	976993	0	50	**8.814**	961296	0	50
Ratio	1.000	1.000			1.007	1.069			0.968	1.073		

Table 3. A comparison of Capo with ROOSTER extensions to Cadence AmoebaPlace on the IWLS 2005 Benchmarks [20]. All routed wirelengths (rWL) are in meters. "Time" represents routing runtime in minutes. ROOSTER outperforms AmoebaPlace by 12.0% in rWL and 1.1% in via counts. Best rWL and via counts are in bold.

Benchmark	ROOSTER + NanoRoute				AmoebaPlace + NanoRoute			
	rWL	#Vias	#Vio.	Time	rWL	#Vias	#Vio.	Time
aes_core	**1.339**	**125939**	2	32	1.657	131049	1	28
ethernet	**7.287**	**467777**	1	27	7.745	471800	1	28
mem_ctrl	**1.061**	**87276**	0	22	1.224	90067	0	21
pci_bridge32	**1.336**	**114880**	0	35	1.598	117326	2	35
usb_funct	**0.995**	**84717**	0	19	1.106	85739	0	19
vga_lcd	25.906	1131591	2	57	**25.405**	**1076178**	2	90
Ratio	1.000	1.000			1.120	1.011		

Routing Benchmarks. To show Capo's performance on placement instances with routing information, we show results for the IBMv2 [41], IWLS [20] and Faraday suites of benchmarks [1] in Tables 2, 3 and 4. Capo with ROOSTER extensions consistently produces routable placements with the best published routed wirelength on several benchmarks and best via counts overall.

Mixed-size Benchmarks. To show Capo's performance on difficult mixed-size placement instances, we show results on difficult floorplanning instances identified by the authors of [31]. Comparisons of Capo with other tools on two difficult benchmark suites are shown in Tables 5 and 6. Most other tools are unable to place these benchmarks legally within the time limit, but Capo with SCAMPI extensions completes all of these benchmarks quickly and legally. Considering the designs successfully placed by PATOMA 1.0 and Capo 9.4, Capo with SCAMPI extensions produces placements with smaller HPWL by 31% and 13%.

Table 4. Routing results on the Faraday benchmarks with movable macro blocks fixed [1]. All routed wirelengths (rWL) are in meters. "Time" represents routing runtime in minutes. Best rWL and via counts are highlighted in bold.

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

Table 5. Runs of Capo with SCAMPI extensions and other tools on recent designs from Calypto Design Systems, Inc. [31]. Best legal solutions are emphasized in bold.

cal	PATOMA 1.0			Capo 9.4 -faster			APlace 2.0			FengShui 5.1			SCAMPI				
bench	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	**17.7**	0.0	39.5	0.10x	0.94x
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	**26.9**	0.0	264.4	0.51x	-
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	**2.8**	0.0	11.4	0.99x	-
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	**5.5**	0.0	26.0	0.73x	-
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	**30.3**	0.0	157.2	0.24x	-
542	0.9	0.0	0.1	0.8	0.0	3.3	0.7	0.1	42.8	×	×	×	**0.8**	0.0	2.0	0.85x	0.96x
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	**71.6**	0.0	188.5	0.86x	-
583	47.0	0.0	2.3	26.1	0.6	190.6	20.6	0.2	421.2	×	×	×	**21.5**	0.0	141.3	0.46x	-
588	8.8	0.0	0.7	6.3	1.1	60.4	4.8	0.5	41.5	×	×	×	**5.6**	0.0	26.4	0.63x	-
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	**3.4**	0.0	11.5	0.68x	-
DCT	×	×	×	×	×	>1800	33.1	1.7 ⊗	719.4	184.7	0.0	8.0	**37.2**	0.0	123.5	-	-
													Average			0.51x	0.95x

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

ISPD Contest Benchmarks. The ISPD 2005 and 2006 Placement Contests introduced sixteen new benchmarks into the public domain based on industrial designs. These designs have many movable objects, an abundance of fixed obstacles and relatively low utilizations. Tables 7 and 8 compare Capo's performance at the contests to Capo's current performance on the 2005 and 2006 contest benchmarks, respectively. Since the contests, Capo has been able to improve its solution quality by 3.1% on the ISPD 2005 benchmarks (while using considerably less runtime than the week allowed for the original contest) and 6.3% for the ISPD 2006 benchmarks.

Since the ISPD 2005 and 2006 contests, variants of the contest benchmarks have been proposed with known optimal or near-optimal wirelengths in order to gauge how much room for improvement is left with state-of-the-are placement methods. In the original work on placements with known optimal solutions, Capo 8.6 placements had nearly twice the HPWL of optimal placements [16]. As shown in Table 9, Capo placements are less than 60% from optimal which represents a significant improvement, especially on such challenging benchmarks as the ISPD 2005 contest benchmarks. The focus of the ISPD 2006 benchmarks is less on HPWL and more on satisfying the required density constraints, and on these benchmarks Capo achieves the density constraint requirements but, as Table 10 shows, at the expense of HPWL were Capo produces solutions with more than twice the optimal wirelength on average.

Table 6. Runs of Capo with SCAMPI extensions and other tools on the IBM-HB$^+$ benchmarks [31]. Best legal solutions are emphasized in bold.

ibm	PATOMA 1.0			Capo 9.4 -faster			APlace 2.0			FengShui 5.1			SCAMPI				
-HB$^+$	HPWL	ovlp	time	HPWL	ovlp	time	HPWL	ovlp	time	HPWL	ovlp	time	HPWL	ovlp	time	*vs.* PATOMA	*vs.* CAPO
bench	(e+06)	(%)	(s)	(e+06)	(%)	(s)	(e+06)	(%)	(s)	(e+06)	(%)	(s)	(e+06)	(%)	(s)	(HPWL)	(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	**3.2**	0.0	57.6	0.83x	-
02	×	×	×	19.1	0.0	1539.7	5.0	2.6	101.5	8.7	0.9 ⊗	43.6	**6.9**	0.0	185.4	-	0.36x
03	×	×	×	×	×	>1800	7.4	2.1	101.3	×	×	×	**10.1**	0.0	179.9	-	-
04	×	×	×	×	×	>1800	8.2	2.8	113.9	10.8	0.2 ⊗	41.4	**11.1**	0.0	145.8	-	-
06	×	×	×	×	×	>1800	8.2	1.0	122.5	10.7	1.4 ⊗	36.0	**9.3**	0.0	201.7	-	-
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	16.1	0.0	90.7	0.96x	1.02x
08	×	×	×	×	×	>1800	16.6	1.0 ⊗	294.2	21.8	0.5 ⊗	60.6	**18.8**	0.0	240.0	-	-
09	×	×	×	20.2	0.2	188.9	15.1	0.9	222.4	20.6	1.2 ⊗	42.9	**20.9**	0.0	185.7	-	-
10	×	×	×	45.9	2.7	263.7	39.9	0.4	544.8	×	×	×	**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	26.9	0.0	137.3	1.06x	0.96x
12	×	×	×	**63.4**	0.0	482.2	×	×	>1800	52.3	0.0 ⊗	39.2	64.0	0.0	397.6	-	1.01x
13	**37.5**	0.0	34.7	39.6	0.0	221.5	31.7	0.5	240.4	×	×	×	39.7	0.0	159.8	1.06x	1.00x
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	**63.8**	0.0	238.8	0.93x	0.94x
15	×	×	×	×	×	>1800	87.5	1.5	422.2	90.6	0.0 ⊗	100.3	**86.4**	0.0	508.3	-	-
16	**100.3**	0.0	74.4	106.9	0.0	431.5	89.8	0.3	528.1	×	×	×	101.8	0.0	254.2	1.01x	0.95x
17	**141.4**	0.0	95.9	152.6	0.1	397.1	133.9	0.5	799.3	×	×	×	146.3	0.0	380.0	1.03x	-
18	**72.6**	0.0	67.2	75.9	0.7	220.1	69.1	0.6	344.0	×	×	×	74.7	0.0	181.9	1.03x	-
													Average			0.99x	0.85x

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

Table 7. Comparison of Capo's current performance to that at the ISPD 2005 Placement Contest. Capo was run using the commandline options "-ispd05" and "-tryHarder". The results for Capo at the ISPD 2005 Placement Contest were the best placements produced over the period of 1 week. Current Capo results are the best of three independent runs of Capo.

Benchmark	ISPD 2005	Current		
	HPWL (e8)	HPWL (e8)	Runtime (m)	HPWL Ratio
adaptec1	-	0.863	95	-
adaptec2	0.997	1.001	128	1.004
adaptec3	-	2.340	274	-
adaptec4	2.113	2.071	257	0.980
bigblue1	1.082	1.071	152	0.990
bigblue2	1.723	1.624	291	0.943
bigblue3	3.826	4.006	984	1.047
bigblue4	10.988	9.470	1335	0.862
Average				0.969

11 Conclusions

In this chapter, we have described in detail the workings of the robust and scalable academic placement tool Capo. Capo is a min-cut floorplacer that provides (i) scalable multi-way partitioning, (ii) routable standard-cell placement, (iii) integrated mixed-size placement, (iv) wirelength-driven fixed-outline floorplanning as well as (v) incremental placement. Capo produces best published results on several publicly available benchmark suites for routability as well as difficult instances of floorplacement. Capo has been used as part of Synplicity's Amplify ASIC product and is freely available for all uses as part of the UMpack (`http://vlsicad.eecs.umich.edu/BK/PDtools/`).

Table 8. Comparison of Capo's current performance to that at the ISPD 2006 Placement Contest. "Overflow" represents the HPWL penalty for not effectively enforcing density constraints on the benchmarks. Results at the ISPD06 contest were the result of a single run of Capo. Current results are the median of three independent runs of Capo. Using the SCAMPI improvements, Capo's HPWL is reduced by 6.3% overall.

Benchmark	ISPD 2006			Current			
	HPWL (e8)	Over-flow%	Runtime (m)	HPWL (e8)	Over-flow%	Runtime (m)	HPWL Ratio
adaptec5	4.916	0.62	162	4.836	0.42	153	0.984
newblue1	0.984	0.13	43	0.850	0.12	47	0.864
newblue2	3.086	0.29	94	2.866	0.21	125	0.929
newblue3	3.612	0.01	101	3.299	0.01	92	0.913
newblue4	3.583	1.15	115	3.512	0.83	96	0.980
newblue5	6.574	0.33	348	6.391	0.26	212	0.972
newblue6	6.683	0.05	308	6.522	0.05	251	0.976
newblue7	15.185	0.02	916	13.482	0.01	525	0.888
Average							0.937

Table 9. Comparison of Capo's current performance on the PEKO-ISPD 2005 benchmarks to optimal results. Capo results are the best of three independent runs of Capo. These results represent an improvement in Capo's performance vs. optimal since the original work on placements with know optimal solutions where Capo placements had nearly twice optimal wire length [16].

Benchmark	Optimal	Capo		
	HPWL (e8)	HPWL (e8)	Runtime (m)	HPWL Ratio
adaptec1	0.201	0.301	35	1.498
adaptec2	0.250	0.401	42	1.604
adaptec3	0.410	0.657	376	1.602
adaptec4	0.394	0.578	455	1.467
bigblue1	0.209	0.296	51	1.416
bigblue2	0.423	0.664	141	1.570
bigblue3	0.944	1.898	321	2.011
bigblue4	1.714	2.533	889	1.478
Average				1.572

References

1. Adya SN, Chaturvedi S, Roy JA, Papa DA and Markov IL (2004) Unification of partitioning, placement and floorplanning. In Proc ICCAD 550–557
2. Adya SN, Markov IL (2003) Fixed-outline floorplanning: enabling hierarchical design. IEEE Trans on VLSI 11(6):1120–1135
3. Adya SN, Markov IL (2005) Combinatorial techniques for mixed-size placement. ACM Trans on Design Auto of Elec Sys 10(5)
4. Adya SN, Markov IL, Villarrubia PG (2006) On whitespace and stability in physical synthesis. Integration: the VLSI Journal 25(4):340–362

Table 10. Comparison of Capo's current performance on the PEKO-ISPD 2006 benchmarks to optimal results. Capo results are the best of three independent runs of Capo.

Benchmark	Optimal	Capo			
	HPWL (e8)	HPWL (e8)	Over-flow%	Runtime (m)	HPWL Ratio
adaptec5	0.611	1.295	4.97	322	2.119
newblue1	0.195	0.563	1.53	29	2.887
newblue2	0.273	0.910	1.17	46	3.333
newblue3	0.303	1.210	1.72	136	3.993
newblue4	0.436	0.792	6.43	196	1.817
newblue5	0.858	1.679	6.33	615	1.957
newblue6	0.800	1.952	2.30	578	2.440
newblue7	1.510	4.196	2.11	1439	2.779
Average					2.580

5. Agnihotri A et al. (2003) Fractional cut: improved recursive bisection placement. In Proc ICCAD 307–310
6. Alpert CJ, Nam G-J, Villarrubia PG, (2003) Effective free space management for cut-based placement via analytical constraint generation. IEEE Trans on CAD 22(10):1343–1353
7. Brenner U, Vygen J (2000) Faster optimal single-row placement with fixed ordering. In Proc DATE 117–121
8. Brenner U, Rohe A (2003) An effective congestion driven placement framework. IEEE Trans. on CAD 22(4):387–394
9. Caldwell AE, Kahng AB, Mantik S, Markov IL, Zelikovsky A (1999) On wirelength estimations for row-based placement. IEEE Trans on CAD 18(9):1265–1278
10. Caldwell AE, Kahng AB, Markov IL (2000) Improved algorithms for hypergraph bipartitioning. In Proc ASPDAC 661–666
11. Caldwell AE, Kahng AB, Markov IL (2000) Can recursive bisection alone produce routable placements? In Proc DAC 477–482
12. Caldwell AE, Kahng AB, Markov IL (2000) Design and implementation of move-based heuristics for VLSI hypergraph partitioning. ACM Journ of Experimental Algorithms 5
13. Caldwell AE, Kahng AB, Markov IL (2000) Optimal partitioners and end-case placers for standard-cell layout. IEEE Trans on CAD 19(11):1304–1314
14. Caldwell AE, Kahng AB, Markov IL. VLSI CAD bookshelf. `http://vlsicad.eecs.umich.edu/BK/`. See also Caldwell AE, Kahng AB, Markov IL (2002) Toward CAD-IP reuse: the MARCO GSRC bookshelf of fundamental CAD algorithms. IEEE Design and Test 72–81
15. Caldwell AE, Kahng AB, Markov IL (2003) Hierarchical whitespace allocation in top-down placement. IEEE Trans on CAD 22(11):716–724
16. Chang C-C, Cong J, Romesis M, Xie M (2004) Optimality and scalability study of existing placement algorithms. IEEE Trans on CAD 23(4):537–549
17. Chen TC, Chang YW, Lin SC (2005) IMF: interconnect-driven multilevel floorplanning for large-scale building-module designs. In Proc ICCAD 159–164
18. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions. In Proc DAC 175–181
19. Goldberg AV (1997) An efficient implementation of a scaling minimum-cost flow algorithm. ACM J. Algorithms 22:1–29

20. IWLS 2005 Benchmarks, http://iwls.org/iwls2005/benchmarks.html
21. Kahng AB, Mantik S, Markov IL, (2002) Min-max placement for large-scale timing optimization. In Proc ISPD 143–148
22. Kahng AB, Mandoiu II, Zelikovsky A (2003) Highly Scalable Algorithms for rectilinear and octilinear steiner trees. In Proc ASPDAC 827–833
23. Kahng AB, Wang Q (2005) Implementation and extensibility of an analytic placer. IEEE Trans on CAD 25(5):734–747
24. Kahng AB, Reda S (2004) Placement feedback: a concept and method for better min-cut placement. In Proc DAC 143–148
25. Karypis G, Aggarwal R, Kumar V, Shekhar S (1997) Multilevel hypergraph partitioning: applications in VLSI domain. In Proc DAC 526–629
26. Khatkhate A, Li C, Agnihotri AR, Yildiz MC, Ono S, Koh C-K, Madden PH (2004). Recursive bisection based mixed block placement. In Proc ISPD 84–89
27. Li C, Xie M, Koh C-K, Cong J, Madden PH (2004) Routability-driven placement and white space allocation. In Proc ICCAD 394–401
28. Li C, Koh C-K, Madden PH (2005) Floorplan management: incremental placement for gate sizing and buffer insertion. In Proc ASPDAC 349–354
29. Moffitt MD, Ng AN, Markov IL, Pollack ME (2006) Constraint-driven floorplan repair. In Proc DAC 1103–1108
30. Nam G-J, Alpert CJ, Villarrubia P, Winter B, Yildiz M (2005) The ISPD 2005 placement contest and benchmark suite. In Proc ISPD 216–220
31. Ng AN, Markov IL, Aggarwal R, Ramachandran V (2006) Solving hard instances of floorplacement. In Proc ISPD 170–177
32. Papa DA, Adya SN, Markov IL (2004) Constructive benchmarking for placement. In Proc GLSVLSI 113–118 http://vlsicad.eecs.umich.edu/BK/FEATURE/
33. Reda S, Chowdhary A (2006) Effective linear programming based placement methods. In Proc ISPD 186–191
34. Roy JA, Adya SN, Papa DA, Markov IL (2006) Min-cut floorplacement. IEEE Trans on CAD 25(7):1313–1326
35. Roy JA, Markov IL (2007) Seeing the forest and the trees: Steiner wirelength optimization in placement. IEEE Trans on CAD 26(4):632–644
36. Roy JA, Markov IL (2007) ECO-system: embracing the change in placement. In Proc ASPDAC 147–152
37. Roy JA, Papa DA, Adya SN, Chan HH, Lu JF, Ng AN, Markov IL (2005) Capo: robust and scalable open-source min-cut floorplacer. In Proc ISPD 224–227
38. Roy JA, Papa DA, Ng AN, Markov IL (2006) Satisfying whitespace requirements in top-down placement. In Proc ISPD 206–208
39. Tang X, Tian R, Wong MDF (2005) Optimal redistribution of white space for wire length minimization. In Proc ASPDAC 412–417
40. Westra J, Bartels C, Groeneveld P (2004) Probabilistic congestion prediction. In Proc ISPD 204–209
41. Yang X, Choi B-K, Sarrafzadeh M (2002) Routability driven white space allocation for fixed-die standard-cell placement. IEEE Trans on CAD 22(4):410–419