Optimal End-Case Partitioners and Placers for Standard-Cell Layout

A. E. Caldwell, A. B. Kahng and I. L. Markov

UCLA Computer Science Dept., Los Angeles, CA 90095-1596 USA

Abstract

We develop new optimal partitioning and placement codes for end-case processing in top-down standard-cell placement. Such codes are based on either enumeration or branch-and-bound, and are invoked for instances below prescribed size thresholds (e.g., < 30 cells for partitioning, or < 10 cells for placement). Our optimal partitioners handle tight balance constraints and uneven cell sizes transparently, while achieving substantial speedups over single FM starts. Optimal cutsizes for small instances (between 10 and 35 movable nodes) are typically found to be at least 40% smaller than what FM will achieve in several starts. Our optimal placers use branch-and-bound to achieve substantial speedups over even Gray code based enumeration. In the context of a top-down global placer, the right combination of optimal partitioners and placers can achieve up to an average of 10% wirelength reduction and 50% CPU time savings for a set of industry testcases. The paper concludes with directions for future research.

1 Introduction

In the placement phase of physical design for standard-cell VLSI circuits, the essential components of a given placement problem are the *placement region*, possibly with discrete allowed locations, the *modules* that are to be placed subject to various constraints, and the *netlist topology* that shapes the objective function being minimized. Commercial standard-cell placers typically apply a top-down, divide-and-conquer approach to define an initial *global placement*. The top-down approach seeks to decompose the given placement problem into smaller problems by subdividing the placement region, assigning modules to subregions, reformulating constraints, and cutting the netlist – such that good solutions to subproblems combine into good solutions of the original problem.

In practice, the problem decomposition is accomplished by hypergraph partitioning. Each hypergraph bipartitioning instance is induced from a rectangular region, or *block*, in the layout:[1] nodes correspond to cells inside the block as well as propagated external terminals [6], and hyperedges are induced over the node set from the original netlist. The actual hypergraph partitioning is performed using FM-type iterative partitioning heuristics with minimum net cut objective [12, 9]; the multilevel paradigm can be applied for larger instances [3, 11]. After a global placement solution has been found (a minimum requirement being that all cells are placed at legal sites in cell rows, with no overlaps),

[1]A *block* conceptually corresponds to (i) a placement region with allowed locations, (ii) a collection of modules to be placed in this region, (iii) all nets incident to the modules, and (iv) locations of all modules beyond the given region that are adjacent to the modules to be placed in the region (such external modules are considered to be terminals for the block, and their locations are fixed).

```
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
```

Figure 1: High-level outline of the top-down partitioning-based placement process.

detailed placement refinement can occur.[2] A high-level pseudocode for top-down bipartitioning-based global placement is shown in Figure 1.

Several unique characteristics of the bipartitioning instances are due to the placement process. In particular, tight *balance constraints* are imposed, i.e., the sizes of partitions in the solution are not allowed to deviate from target partition sizes (see [4] for a review of netlist partitioning formulations and constraints). Such constraints arise because the proportion of free sites ("whitespace") in n-layer metal deep-submicron designs is typically less than a few percent; hence, total total module area assigned to a block must closely match the available layout area in the block. When blocks are partitioned by horizontal cutlines, the discrete row structure of the layout also forces tight balance tolerances. Although the location of vertical cutlines may enjoy slightly more flexibility, the difficulty of managing terminal propagation, block definition, region-based wirelength estimation, etc. again precludes the use of large balance tolerances. Essentially, relaxed balance tolerances can lead to uneven area utilization and overlapping placements.

As shown in Figure 1, when the partitioning instance is sufficiently small or has sufficiently large block aspect ratio (e.g., when the block has only one cell row), *end-case processing* is applied in the form of an alternate partitioner or a placer. For example, an instance of four cells will not be recursively bipartitioned. Rather, the four cells will be placed optimally, e.g., by exhaustive enumeration of all $24 = 4!$ placements to find the best one. Of course, due to the combinatorial nature of the problem, it is not feasible to apply optimal algorithms to even moderately large partitioning and placement instances. Factors such as initialization overhead (e.g., building gain bucket structures in the FM algorithm), solution quality, and runtime together determine the problem size at which it is best to switch over from the default (FM-based) hypergraph bipartitioner to a given end-case

[2] The authors of [2] note that the "quadratic placement methodology" also fits this model, in that quadratic placers still employ hypergraph partitioning, but with initial partitioning solutions obtained from analytic placements (cf. PROUD [20] or GORDIAN [13]).

algorithm.

1.1 Motivations for Optimal End-Case Processing

With each new deep-submicron process generation, there is a wider range of cell sizes in cell libraries. For example, an 80x range of buffer strengths is not uncommon today, and the number of complex gates in the library also increases. This is due to the wider range of interconnect layer RC parameters, and to new methodologies for achieving performance convergence via sizing-based optimizations [14, 15]. In the context of tight partitioning area balance constraints, the increased variation in cell sizes leads to more difficult instances for FM-based partitioners. Such partitioners are less likely to give high-quality results because (i) the FM algorithm may never reach the feasible part of the solution space (especially if it has trouble finding an initial balance-feasible solution), and (ii) even a relative scarcity of feasible moves (from any given feasible solution) can make the algorithm more susceptible to being trapped in a bad local minimum (cf. the analysis of Dutt and Theny [8]).

Even if the partitioning instance does not have a "tight" balance constraint, it is not clear whether traditional FM-based algorithms will yield good solution quality. As discussed in the Rent's rule based wirelength estimation literature (e.g., [18] [5]), any suboptimality in cutsize for a given bipartitioning instance will tend to increase both the number of terminals in later bipartitioning instances and the total wirelength of the placement. Pathological examples for the FM algorithm are easy to construct,[3] and the pitfalls of the recursive bisection approach are well-known [17]. Yet, to our knowledge there is no work in the literature that quantifies the suboptimality of the FM algorithm in practice, except for large "self-scaled" instances [10].[4] At the same time, many small bipartitioning instances are created during the course of top-down placement, and their solutions contribute significantly to the overall wirelength of the global placement solution. Moreover, current implementations of global placement, to our knowledge, still employ FM-based heuristics even for relatively small instances. It is natural to ask whether there can be any benefit from improved bipartitioning methods, if only for smaller instances.

[3] A 12-node, 14-edge example has nodes A_i, B_i, C_i, D_i for $i = 1, 2, 3$, and edges forming cliques over the A's, the B's, the C's and the D's, along with an A_1-C_1 edge and a B_1-D_1 edge. The cliques over the B's and D's have weight 2 per edge; all other edges have weight 1. All nodes have weight 1, and the balance constraint is for exact bisection. Suppose the initial solution has all A's and B's in Partition 0, and all C's and D's in Partition 1 (i.e., cutsize = 2). Then, the first FM pass will move $A_1, C_2, A_2, C_3, A_3, C_1, B_1, D_2, B_2, D_3, B_3, D_1$ in that order, and FM will then terminate. However, the optimal cutsize is 0.

[4] We have found one public-domain code that provides an optimal partitioner, namely, the graph partitioning package PARTY [16]. This code deals only with graphs, and thus cannot be used for VLSI instances. We have examined the source code in detail, and have determined that it strongly exploits the fact that the input is a graph (i.e., "all nets have exactly two pins"). Adapting PARTY code to the VLSI context is therefore not feasible.

Given these motivations, our present work studies the potential benefits of "improved" bipartitioning methods, specifically focusing on *optimal* partitioners that are based on enumeration or branch-and-bound. We also study linear placement for end-case processing, again focusing on optimal methods. The goals of this research are to (i) to assess the cutsize suboptimality of traditional FM-based approaches for small partitioning instances arising in top-down placement, (ii) assess the runtime penalty that can also be incurred with traditional FM-based approaches, and (iii) determine the overall effect of new "end-case placers and partitioners" in a generic top-down placer implementation.

1.2 Contributions and Organization of Paper

In this paper, we develop new, optimal "end-case partitioners" and "end-case placers" for end-case processing in top-down layout. We explore the tradeoffs between (i) exhaustive enumeration approaches (based on either Gray code or lexicographic orderings) and (ii) branch-and-bound approaches; we also give insights to guide efficient implementations. Section 2 and the Appendix describe the implementation of optimal partitioning algorithms. We compare our implementations against LIFO- and CLIP-FM [7] for suites of small partitioning instances that arise during the top-down placement of industry standard-cell designs. The experimental data shows that our end-case partitioners enjoy runtime advantages over both LIFO- and CLIP-FM for surprisingly large instance sizes, while also yielding significantly improved solution qualities. Section 3 and the Appendix describe the implementation of optimal linear placement algorithms. Section 4 evaluates the impact of optimal partitioning and placement on a top-down global placer. We provide details of the top-down placer, followed by experimental data showing that using the right combination of optimal partitioners and placers can achieve up to an average of 10% wirelength reduction while producing up to a 50% CPU time savings for a set of industry testcases, when compared against using traditional FM-based partitioners.

2 End-Case Partitioning

We have explored two optimal algorithms for small instances of hypergraph partitioning: Gray code based enumeration, and branch-and-bound.

- A *Gray code ordering* traverses all partitioning solutions using single-node partition-to-partition moves; this is attractive for exhaustive enumeration because updating cutsize between successive solutions does not require much runtime. (Updating the cutsize of a new solution requires updating only the cut of each net incident to the moved node.)

- Branch-and-bound performs depth-first traversal of a tree of *partial partitioning solutions*, i.e., assignments of some nodes to partitions. A root-leaf path in this tree will construct a partitioning solution, one node assignment at a time. With each node assignment, a lower bound on the cutsize can be updated, and will converge to the actual cutsize of a complete solution when the leaf vertex is reached. If a solution with cutsize c_0 has already been found, the algorithm will not consider any extensions of a partial solution whose lower bound on cost is $\geq c_0$. This is because such extensions cannot lead to a better solution, and the subtree of such extensions is *bounded* from consideration. We observe that without bounding, branch-and-bound would simply perform lexicographic enumeration of solutions, which is likely to be less efficient than Gray code based enumeration. In the lexicographic ordering of complete partitioning solutions of N nodes, $\Theta(N)$ partition reassignments are required on average between successive solutions. Thus, effective bounding is necessary for branch-and-bound to be faster than Gray code based enumeration.

2.1 Gray Codes

Gray code enumeration starts with the partitioning solution that assigns all nodes to partition zero, and reassigns one node at a time with each reassignment producing a solution never seen before. The sequence of solutions can be interpreted as a space-filling curve in the space of partitioning solutions (e.g., the space of 2-way partitionings of N nodes is the set of corners of N-cube).

We represent a Gray code for bipartitioning N nodes as a *Gray sequence* of $2^N - 1$ numbers taken from the set $\{0, \ldots, N-1\}$. These numbers are interpreted as instructions to reassign the respective nodes to the "other" partition. For example, the Gray sequence for bipartitionings of 1 item is just { `0` }, the sequence for bipartitionings of 2 items is { `0 1 0` }, and the sequence for 3 items is { `0 1 0 2 0 1 0`}. A Gray sequence for k-way partitioning will have a sequence of $k^N - 1$ numbers, each interpreted as reassignment of the given node to the next higher partition index (modulo k). The corresponding recursive construction is implemented by the following optimized C++ code, in which `numPart` denotes the number of partitions, and `size` is the number of nodes in the partitioning instance.

```
byte* begin=_tables[size];                          // e.g., typedef byte char;
byte* ptr  = begin;
for(unsigned p=numPart-1; p!=0; p--)  *ptr++=0; // initialize recursion
```

```
for(unsigned i=1; i!=size; i++)
{
  unsigned bytesToCopy=ptr-begin;
  for(p=numPart-1; p!=0; p--)
  {
       *ptr++=i;
       memcpy(ptr,begin,bytesToCopy);
       ptr+=bytesToCopy;
  }
}
```

Our Gray code based enumerative partitioner incrementally computes partition balances and cuts for each solution it sees. If a solution is better than the best seen so far (e.g., satisfies balance constraints and has smaller cut), it is recorded as best. A small speedup can result from having a lower bound for solution cost (e.g., 0 is always a valid bound for the net cut objective), since the partitioner can return once a solution with that cost is found. Also, straightforward extensions are available in the case when a legal solution is not guaranteed, e.g., the best balanced solution can be found with cut-based tiebreaking, or a quick check for legal solutions can be performed before a full-fledged pass through the Gray sequence with incremental cut computation.

2.2 Branch-and-Bound

The key observation underlying branch-and-bound is that a lower bound for net cut, "cut so far", is available given assignments of only some nodes. Namely, a hyperedge is considered "already cut" if it has two nodes assigned to different partitions, and "uncut so far" otherwise. A similar observation applies to partition balances. All nodes are ordered from the start, with fixed nodes (i.e., terminals) followed by movable (i.e., assignable) nodes. A given node $i > 0$ can be assigned to a partition only after node $i-1$ has been assigned. Our implementation sorts the movable nodes in ascending order of degree, in order to promote more efficient bounding.

Figures 3 and 4 in the Appendix give fairly detailed pseudocode for branch-and-bound partitioning, accompanied by some implementation notes. The algorithm operates on a "main stack" that (i) stores partition assignments for all nodes assigned so far, and (ii) allows nodes to be "unassigned" in the reverse order of how they were assigned. Because of this structure, no hyperedges have to be traversed: rather, when a node is assigned to a partition without violating balance constraints, all incident "uncut so far" hyperedges are updated. If for a given hyperedge this node is the first assigned node, the hyperedge is marked with the index of the partition to which the node is assigned. Otherwise, the new assignment is compared to previous assignments of nodes on the hyperedge, to check if the net becomes cut (if the net becomes newly cut, the total cut so far is incremented).

Branching is done by pushing a new partition assignment onto the main stack. Bounding is done by popping partition assignments from main stack and is triggered by either partition balances violating prescribed limits or by "cut so far" reaching the cutsize of a previously seen solution. Straightforward extensions are available if the existence of legal balanced solutions is not guaranteed; these are similar to those given for Gray code based enumerative partitioners.

2.3 Comparison of Optimal Partitioning Algorithms

We now assess the speed and solution quality improvements that can be obtained using Gray code enumeration or branch-and-bound partitioners.

Provenance of Small Instances

Our testbed consists of small hypergraph bipartitioning problems saved from our top-down standard-cell placer, which is described in Section 4 below. We have saved all instances with between 10 and 35 (movable) *non-terminal* nodes that arise during the top-down placement of Test Case 1 and Test Case 3, out of the five industrial test cases described in Table 4 below. These small instances have fairly uniform statistical properties across designs that we have seen; typical statistics (for the Test Case 3 small instances) are given in Table 1. We give the number of instances of each size, and the average number of hyperedges, average hyperedge degree, and average node degree for each instance size. We also give the same statistics when only *essential nets* are counted: a net that is guaranteed to be cut in any solution due to fixed terminals is *inessential*, and does not contribute to the runtime of our optimal partitioners.

Runtime Comparisons vs. FM and CLIP

It turns out that Gray code enumeration is competitive with branch-and-bound only for very small instances. We may compare the two optimal approaches using *runtime ratio*, i.e., the ratio of CPU seconds spent on the same problem instances. Instances for which either of the CPU readings is less than 0.0001 second[5] are considered unreliable and are dropped from the test suite. We then compute the geometric mean of the runtime ratios for the remaining "good" instances. Our two implementations perform comparably on instances with 9 modules, with Gray code enumeration being 1.9 times slower on instances with 10 modules. The runtime ratio (Gray code runtime divided by branch-

[5]All of our CPU times are reported for a 300MHz Sun Ultra-10 with 128MB RAM.

No. of	No. of	All Edges			Essential Edges		
NonTerms	Problems	Num Edges	Edge Deg	Node Deg	Num Edges	Edge Deg	Node Deg
10	160	16.87	2.189	3.693	15.11	2.196	3.317
11	145	18.1	2.196	3.612	16.33	2.204	3.272
12	94	19.63	2.215	3.622	17.73	2.223	3.285
13	85	20.52	2.256	3.56	18.66	2.269	3.257
14	58	23.28	2.241	3.727	21.12	2.248	3.392
15	78	25.94	2.244	3.88	23.54	2.252	3.533
16	65	27.72	2.251	3.901	25.06	2.261	3.541
17	68	29.19	2.276	3.908	26.16	2.294	3.53
18	40	32.02	2.291	4.076	28.7	2.3	3.667
19	47	33.02	2.288	3.976	29.36	2.304	3.561
20	42	34.76	2.299	3.995	30.62	2.315	3.544
21	44	36.91	2.302	4.045	32.59	2.321	3.602
22	27	39.81	2.264	4.098	35.56	2.27	3.668
23	37	40.43	2.338	4.109	36.54	2.335	3.71
24	30	40.83	2.286	3.889	35.97	2.304	3.453
25	32	42.56	2.33	3.966	37.84	2.35	3.558
26	38	44.08	2.349	3.983	40	2.349	3.613
27	34	44.94	2.366	3.938	40.12	2.389	3.549
28	31	47.13	2.337	3.933	41.71	2.357	3.51
29	21	49.1	2.346	3.972	44.57	2.359	3.626
30	25	50	2.41	4.016	44.8	2.417	3.609
31	12	48.75	2.356	3.704	43.33	2.377	3.323
32	13	51.69	2.369	3.827	46.69	2.39	3.488
33	9	49.78	2.342	3.532	44	2.341	3.121
34	13	53.62	2.31	3.643	47.77	2.337	3.283
35	9	54	2.465	3.803	49.11	2.475	3.473

Table 1: Statistics of end-case problem instances for Test Case 3. We also show the same statistics for *essential edges* only (i.e., omitting edges that are guaranteed to be cut in any partitioning.

and-bound runtime) increases by a factor of between 1.5 and 1.9 for each additional module. Thus, we have compared only our branch-and-bound code against the LIFO FM and CLIP [7] algorithms. (While the Gray code enumeration is faster for instances of 8 modules or less, but such instances are better handled by the end-case placers described in Section 3.)

To compare the FM heuristic to branch-and-bound, we must account for randomization and the fact that FM does not always achieve optimal solutions. For each instance in our test suite, our experiments record the average cutsize achieved by one start of FM, as well as the average best cutsize achieved over 2, 3 and 100 starts. Then, after running branch-and-bound on the same instance, we can calculate two figures of merit: the *runtime ratio* (FM runtime divided by branch-and-bound runtime), and the *quality ratio* (average FM cutsize divided by branch-and-bound (i.e., optimal) cutsize). We also compute the analogous figures of merit when 2, 3 or 100 starts of FM are used. All ratios are averaged geometrically over all "good" instances of each size, where "good" excludes instances with optimal cutsize equal to zero, as well as instances that are solved by branch-and-bound in less than 0.0001 second. Finally, we repeat the entire experiment using the CLIP algorithm of Dutt and Deng

[7], which is in general a stronger flat partitioner. We note that our FM implementation is faster, and obtains as good or better solution quality on average, than the public-domain implementation of W. Deng that is available from C. J. Alpert's web page [1]. Our CLIP implementation exhibits similar quality relative to reported implementations.

Experimental results are shown in Tables 2 and 3 for Test Cases 1 and 3. We see that FM is clearly slower than branch-and-bound on all instances of 23 modules or less. This is explained by the relatively high overhead (notably the complicated gain update mechanism) of any FM implementation: during each FM pass a hyperedge of degree p can be traversed p^2 times, while branch-and-bound never traverses hyperedges.

We also see that the solution quality achieved by several starts of FM is considerably worse than the optimal cost. In fact, for many instances FM did not find the optimal cost in 100 starts. The CLIP algorithm in general fared no better. As noted in Section 1, we may distinguish two potential problems for FM on small balanced hypergraph partitioning instances: (i) poor reachability in the solution space due to the balance constraint, and (ii) weakness of the FM neighborhood operator. The former means that not all feasible solutions can be reached from a given solution by legal single-module partition-to-partition moves, while the second problem is more fundamental and can be rephrased as "FM simply makes wrong moves".

To ensure that our test instances are not overconstrained, and thus decrease the likelihood of (i), we set the partitioning tolerance to the maximum of the *average* module area and either 2% or 10% of the total module area, for vertical and horizontal cutlines respectively. The harsher tolerance for horizontal cutlines is dictated by area utilization considerations for neighboring rows; as noted in Section 1, such a constraint is not easily relaxed without incurring module overlaps and uneven resource utilization. However, our top-down algorithm for splitting blocks encourages more horizontal cutlines at earlier stages (see Section 4.1), so that the smaller partitioning instances in our test suite tend to have vertical cutlines and lax partitioning tolerances.

3 End-Case Placement

In the top-down partitioning based placement approach, the original placement problem (considered as a "block") is partitioned into two subproblems (sub-blocks) and then recursively, into smaller and smaller subproblems (see Figure 1). Eventually, blocks containing very few nodes are created for which wirelength can be directly optimized, e.g., by exhaustive search.

TEST CASE 1 / LIFO FM										
Nodes	Instances (good)	Sub opt	1 start		2 starts		3 start		100 starts	
			time	cut	time	cut	time	cut	time	cut
10	24(**20**)	7	23.235	2.035	46.470	1.796	69.704	1.670	2323.480	1.175
11	37(**30**)	6	16.631	2.018	33.262	1.730	49.893	1.591	1663.110	1.064
12	31(**26**)	1	17.246	2.291	34.491	1.961	51.737	1.799	1724.570	1.020
13	22(**19**)	2	22.650	2.199	45.300	1.867	67.950	1.711	2265.000	1.029
14	22(**21**)	7	14.292	2.037	28.585	1.766	42.877	1.639	1429.240	1.069
15	20(**20**)	4	11.461	2.001	22.921	1.734	34.382	1.617	1146.060	1.056
16	9(**9**)	4	10.133	1.690	20.267	1.493	30.400	1.404	1013.340	1.095
17	12(**11**)	4	7.066	1.887	14.132	1.677	21.198	1.579	706.615	1.077
18	6(**6**)	5	8.281	1.915	16.561	1.722	24.842	1.639	828.053	1.256
19	8(**8**)	5	9.344	2.315	18.687	2.007	28.031	1.857	934.355	1.173
20	11(**11**)	9	3.562	2.340	7.124	2.088	10.687	1.959	356.222	1.275
21	12(**12**)	10	3.361	2.258	6.723	2.027	10.084	1.916	336.142	1.257
22	10(**10**)	9	3.831	2.099	7.662	1.904	11.493	1.800	383.102	1.242
23	7(**7**)	7	1.312	2.166	2.624	1.979	3.936	1.884	131.201	1.371
24	8(**8**)	8	1.187	2.154	2.373	1.968	3.560	1.877	118.650	1.394
25	7(**7**)	7	**1.325**	2.342	2.651	2.114	3.976	2.003	132.543	1.403
26	11(**11**)	11	0.703	2.473	1.405	2.266	2.108	2.158	70.259	1.503
27	8(**8**)	8	0.662	2.405	1.324	2.183	1.986	2.083	66.189	1.482
28	10(**10**)	10	0.418	2.522	0.835	2.286	1.253	2.168	41.771	1.403
29	9(**9**)	9	0.746	2.316	1.492	2.118	2.238	2.019	74.595	1.434
30	2(**2**)	2	1.026	3.094	2.052	2.803	3.078	2.654	102.588	1.789
31	7(**7**)	4	0.596	1.958	1.192	1.811	1.788	1.743	59.599	1.474
32	4(**4**)	2	0.675	2.196	**1.351**	1.930	**2.026**	1.800	67.532	1.273
33	1(**1**)	1	0.213	3.046	0.427	2.801	0.640	2.700	21.333	2.143
34	3(**3**)	3	0.142	2.453	0.285	2.258	0.427	2.151	14.231	1.641
35	2(**2**)	2	0.007	2.062	0.014	1.932	0.021	1.854	0.707	1.401
TEST CASE 1 / CLIP FM										
10	24(**20**)	9	24.083	1.938	48.166	1.710	72.248	1.600	2408.280	1.180
11	37(**31**)	7	22.605	1.992	45.209	1.692	67.814	1.552	2260.460	1.057
12	31(**26**)	3	18.949	2.134	37.899	1.839	56.848	1.700	1894.930	1.040
13	22(**17**)	6	18.002	2.248	36.005	1.910	54.007	1.762	1800.230	1.105
14	22(**19**)	4	15.056	2.085	30.113	1.802	45.169	1.667	1505.650	1.046
15	20(**20**)	7	14.950	2.018	29.899	1.749	44.849	1.628	1494.950	1.091
16	9(**9**)	2	10.092	1.709	20.184	1.507	30.276	1.420	1009.200	1.026
17	12(**12**)	4	6.797	1.851	13.595	1.648	20.392	1.549	679.742	1.072
18	6(**5**)	4	7.477	1.972	14.953	1.788	22.430	1.702	747.666	1.200
19	8(**8**)	5	8.437	2.335	16.875	2.014	25.312	1.862	843.726	1.218
20	11(**9**)	7	3.683	2.385	7.366	2.130	11.049	1.992	368.310	1.227
21	12(**12**)	11	3.882	2.270	7.764	2.038	11.646	1.922	388.190	1.269
22	10(**10**)	9	2.717	2.117	5.433	1.920	8.150	1.827	271.652	1.285
23	7(**7**)	7	1.316	2.158	2.633	1.964	3.949	1.867	131.645	1.354
24	8(**8**)	7	1.334	2.126	2.668	1.941	4.001	1.851	133.382	1.321
25	7(**7**)	6	**1.387**	2.359	2.775	2.114	4.162	1.997	138.750	1.283
26	11(**11**)	11	0.618	2.461	1.236	2.253	1.853	2.149	61.777	1.497
27	8(**8**)	7	0.544	2.406	1.089	2.171	1.633	2.062	54.444	1.370
28	10(**10**)	10	0.389	2.527	0.778	2.305	1.167	2.199	38.914	1.671
29	9(**9**)	9	0.792	2.320	1.583	2.116	2.375	2.018	79.153	1.394
30	2(**2**)	2	1.772	3.049	3.543	2.807	5.315	2.695	177.157	1.891
31	7(**7**)	4	0.624	1.930	1.247	1.788	1.871	1.734	62.363	1.393
32	4(**4**)	2	0.921	2.206	**1.842**	1.982	**2.763**	1.878	92.094	1.185
33	1(**1**)	1	0.217	3.021	0.433	2.778	0.650	2.672	21.667	2.000
34	3(**3**)	3	0.120	2.464	0.241	2.280	0.361	2.179	12.029	1.689
35	2(**2**)	2	0.007	2.074	0.015	1.932	0.022	1.866	0.731	1.477

Table 2: Comparison of LIFO FM and CLIP FM against Branch-and-Bound, using runtime and solution quality ratios for average of 1 start, average best of 2 starts, average best of 3 starts, and best of 100 starts. Ratios greater than 1.0 indicate FM losses. Transition points for runtime are shown in bold.

TEST CASE 3 / LIFO FM										
Nodes	Instances (good)	Sub opt	1 start time	1 start cut	2 starts time	2 starts cut	3 start time	3 start cut	100 starts time	100 starts cut
10	160(**134**)	32	20.731	1.976	41.463	1.700	62.194	1.564	2073.140	1.080
11	145(**130**)	25	18.847	2.112	37.695	1.803	56.542	1.651	1884.730	1.069
12	94(**83**)	8	17.028	1.948	34.055	1.671	51.083	1.537	1702.760	1.029
13	85(**81**)	10	16.108	2.054	32.216	1.757	48.324	1.609	1610.810	1.030
14	58(**55**)	11	11.149	1.892	22.299	1.623	33.448	1.496	1114.930	1.042
15	78(**76**)	24	10.138	1.840	20.275	1.603	30.413	1.496	1013.770	1.059
16	65(**62**)	20	6.796	1.846	13.592	1.634	20.388	1.530	679.601	1.053
17	68(**68**)	32	5.422	1.933	10.844	1.713	16.266	1.611	542.201	1.118
18	40(**40**)	25	4.430	1.907	8.860	1.717	13.290	1.628	443.011	1.149
19	47(**46**)	38	3.577	1.967	7.154	1.775	10.731	1.681	357.716	1.214
20	42(**40**)	29	2.761	1.913	5.523	1.726	8.284	1.635	276.130	1.178
21	44(**44**)	39	2.191	2.000	4.382	1.806	6.573	1.711	219.106	1.228
22	27(**27**)	22	1.429	2.001	2.857	1.810	4.286	1.721	142.859	1.217
23	37(**37**)	36	**1.134**	1.969	2.268	1.806	3.402	1.721	113.410	1.275
24	30(**30**)	27	0.871	2.088	1.743	1.896	2.614	1.805	87.141	1.294
25	32(**32**)	32	0.826	2.159	1.652	1.993	2.478	1.905	82.607	1.415
26	38(**38**)	38	0.512	2.368	**1.023**	2.171	1.535	2.072	51.163	1.512
27	34(**34**)	31	0.495	2.198	0.990	2.010	1.484	1.913	49.476	1.354
28	31(**31**)	31	0.357	2.227	0.713	2.054	**1.070**	1.963	35.673	1.468
29	21(**21**)	19	0.261	2.201	0.523	2.031	0.784	1.939	26.134	1.434
30	25(**25**)	24	0.151	1.973	0.302	1.834	0.453	1.765	15.110	1.390
31	12(**12**)	10	0.251	2.000	0.502	1.868	0.753	1.805	25.102	1.465
32	13(**13**)	9	0.261	1.698	0.522	1.595	0.783	1.550	26.085	1.287
33	9(**9**)	7	0.106	1.903	0.211	1.782	0.317	1.720	10.560	1.397
34	13(**13**)	13	0.078	2.773	0.155	2.562	0.233	2.447	7.759	1.816
35	9(**9**)	9	0.052	2.326	0.104	2.183	0.157	2.111	5.218	1.678
TEST CASE 3 / CLIP FM										
10	160(**124**)	27	24.238	1.971	48.477	1.688	72.715	1.552	2423.840	1.070
11	145(**120**)	20	21.667	2.129	43.334	1.819	65.000	1.666	2166.680	1.056
12	94(**86**)	9	17.968	1.985	35.937	1.698	53.905	1.563	1796.830	1.035
13	85(**77**)	7	15.763	2.005	31.526	1.712	47.290	1.572	1576.320	1.023
14	58(**55**)	9	10.479	1.867	20.959	1.601	31.438	1.473	1047.940	1.036
15	78(**77**)	24	10.686	1.867	21.372	1.625	32.059	1.508	1068.620	1.068
16	65(**65**)	26	7.488	1.890	14.975	1.670	22.463	1.564	748.765	1.099
17	68(**68**)	35	5.959	1.945	11.918	1.728	17.877	1.623	595.893	1.133
18	40(**40**)	26	3.926	1.908	7.851	1.720	11.777	1.623	392.572	1.157
19	47(**47**)	36	3.481	1.965	6.962	1.774	10.443	1.678	348.104	1.198
20	42(**42**)	29	3.150	1.922	6.301	1.736	9.451	1.645	315.043	1.177
21	44(**43**)	35	2.276	1.989	4.552	1.806	6.827	1.714	227.579	1.213
22	27(**27**)	21	1.422	1.999	2.843	1.817	4.265	1.720	142.166	1.245
23	37(**37**)	34	**1.186**	1.979	2.372	1.813	3.558	1.733	118.593	1.296
24	30(**30**)	29	0.923	2.100	1.846	1.912	2.769	1.818	92.294	1.300
25	32(**32**)	32	0.779	2.151	1.559	1.974	2.338	1.885	77.927	1.398
26	38(**38**)	37	0.519	2.380	1.037	2.185	1.556	2.086	51.867	1.541
27	34(**34**)	33	0.585	2.199	**1.169**	2.008	1.754	1.912	58.457	1.374
28	31(**31**)	31	0.361	2.219	0.723	2.038	**1.084**	1.947	36.133	1.421
29	21(**21**)	20	0.242	2.183	0.485	2.011	0.727	1.925	24.241	1.439
30	25(**25**)	24	0.155	1.988	0.311	1.849	0.466	1.781	15.534	1.369
31	12(**12**)	10	0.248	2.002	0.496	1.865	0.744	1.799	24.807	1.393
32	13(**13**)	9	0.289	1.691	0.578	1.593	0.867	1.554	28.888	1.305
33	9(**9**)	7	0.104	1.913	0.209	1.791	0.313	1.731	10.435	1.374
34	13(**13**)	13	0.080	2.747	0.161	2.540	0.241	2.427	8.049	1.816
35	9(**9**)	9	0.052	2.327	0.105	2.178	0.157	2.103	5.230	1.613

Table 3: Comparison of LIFO FM and CLIP FM against Branch-and-Bound, using runtime and solution quality ratios for average of 1 start, average best of 2 starts, average best of 3 starts, and best of 100 starts. Ratios greater than 1.0 indicate FM losses. Transition points for runtime are shown in bold.

In this section, we describe *end-case placers* that operate on such small problems and produce solutions with minimum half-perimeter wirelength. Our implementation assumes only *single-row* end-case instances, given by:[6]

- A hypergraph with all nodes (cells) having *widths*. The single-row instance implies that all cell heights are assumed to be equal to the row height.
- Every hyperedge has a bounding box of locations of (fixed) terminal pins that the corresponding net has in the original netlist.
- Each hyperedge-to-node connection has a *pin offset* relative to the origin of the respective cell.
- A placement region, i.e., a subrow of a certain length.[7]

An additional requirement, critical for implementations, is that every hyperedge (net) can connect to a node (cell) with at most one pin.

Given this formulation – in particular, the uniform distribution of whitespace – placement solutions become permutations of hypergraph nodes. The end-case placement problem thus naturally lends itself to (i) enumeration via Gray codes, and (ii) branch-and-bound based on lexicographical ordering.

3.1 Gray Code Based Small Placers

With the help of Gray codes, permutations can be enumerated so that each permutation differs from the previous permutation by one transposition of neighboring items [19]. The use of Gray codes is enabled by the fact that swapping two neighboring cells of different widths does not change their sum of widths and the white space between them, and hence does not affect the locations of other cells in the instance.

To find optimal solutions, one needs to traverse all permutations of hypergraph nodes incrementally, updating the total wirelength with every transposition and save the permutation with best-so-far wirelength. The incremental wirelength update is the most critical part of the implementation, and requires a complete traversal of all hyperedges incident to one or both nodes being swapped.[8]

---

[6] This assumption is warranted by the top-down placer implementation described in Section 4.1, which preferentially splits multi-row blocks between rows as the blocks become small.

[7] It may happen that the subrow is too short to accommodate all cells without overlaps. While this is undesirable, an end-case placer handles this by minimizing both wirelength and overlaps. If there is white space, our implementation distributes it evenly.

[8] The necessary incidence information can be produced by a $\Theta(N^2)$ precomputation and stored in $\Theta(N^2)$ space; this is reasonable given that an exponential number of solutions are to be enumerated.

3.2 Branch-and-Bound Based Small Placers

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. If this lower bound is greater than or equal to the best complete solution cost yet found, no extensions of the current partial solution need be considered and the subtree can be bounded away.

One difficulty in applying branch and bound to end-case placement is varying cell widths. Since whitespace is distributed equally between the cells, cells are packed with a fixed-size space between neighboring cells. Replacing the middle cell in a sequence of three with one of different size will force the location of at least one other cell to change; this in turn requires recomputing the bounding boxes of nets attached to the shifted cell(s). To avoid such expense, we use a lexicographic ordering of the permutations. Conceptually, the nodes are packed from left to right. Nodes are always added to or removed from the right end of the (partially-specified) permutation. The lexicographic order of the permutations means that for a given prefix, all placements beginning with that prefix will be visited before the prefix is changed, and none of the cells in the prefix will be shifted. This 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" a node at the end of a partial solution before all lexicographically greater partial solutions have been visited. Figure 4.2 in the Appendix provides pseudocode for our branch-and-bound placer.

4 Optimal End-Case Processing in Global Placement

4.1 Top-Down Placement Testbench

Recall from Figure 1 that, given the concept of placement blocks, top-down placement reduces to only two nontrivial operations: (i) splitting a block, and (ii) solving an endcase. While this paper deals with the latter, specific implementations of the former may have significant effects on features of endcase instances. Thus, we first describe our method of splitting blocks.

Conceptually, a placement block is responsible for the nets (hyperedges) incident to its modules. However, efficient implementations do not have to fully transcribe them from a block to its sub-blocks, because incident nets can be deduced from the original netlist. Each external module of a block (i.e.,

Test Case	Core Cells	Pads	Nets
1	2741	545	3286
2	8829	182	10715
3	11471	662	11673
4	12146	711	10880
5	20392	185	21987

Table 4: Core cell, pad and net counts for test cases used.

a module adjacent to some module in the block, but not itself in the block) is a terminal and is located at the center of the placement region of the block to which it is assigned.

Given such an arrangement, splitting a block reduces to balanced hypergraph partitioning with fixed terminals, as detailed in Figure 4.1. In particular, the possibly numerous terminals of a block will be collapsed into at most two terminals in the corresponding hypergraph bipartitioning instance. Moreover, nets incident to fixed terminals in both partitions become *inessential* (because they will be cut in any partitioning solution) and are therefore removed from consideration.

Our implementation chooses a horizontal cutline to split a block with M modules if the block contains $M/15$ or more rows. Since the blocks are split into sub-blocks as evenly as possible, blocks of size less than 15 cells will typically contain only one row, simplifying endcase analysis.

To assess the impact of end-case partitioners and placers on top-down global placement, we have run the top-down placer described above on 5 industry test cases whose attributes are given in Table 4. For each test case:

- We vary the instance size threshold below which branch-and-bound partitioning is invoked from 0 (i.e., always use FM for partitioning) to 40 (use FM for instances of size greater than 40, and branch-and-bound for instances of size 40 or less). All applications of FM consist of four independent starts; our experience indicates that any smaller number of starts will result in substantial degradation of solution quality, making comparisons uninteresting.

- We vary the size threshold below which the end-case placer is called (i.e., instead of further bipartitioning of the block) from 3 to 8. We report results only for the branch-and-bound end-case placer, since our experiments show that the expense of generating Gray codes for permutations is not justified by the performance of the enumerative placer. In particular, even lexicographic enumeration (i.e. branching without bounding) is typically cheaper than Gray code based enumeration because no hyperedge traversals are required.

Reduction of block splitting to balanced hypergraph partitioning

Input: Original hypergraph with all modules placed at the centers of the placement regions of their blocks;
A collection of modules in the block to be split;
Placement region description for the block to be split (includes legal module locations)

Output: Instance of balanced hypergraph bipartitioning with two partitions and at most two fixed terminals

I. Split the placement region into two subregions (with indices 0 and 1) by vertical or horizontal cutline. (This choice is based on the aspect ratio of the placement region, routing considerations, etc. The subregions will correspond to partitions of the output instance.)

II. Build hypergraph with fixed terminals

1. Create a hypergraph with two terminals vertices 0 and 1, fixed in respective partitions, and a vertex for each movable module in the block
2. for each hyperedge of the original (netlist) hypergraph incident to at least one of the modules in the block:
   (a) clear temporary stack for modules
   termPartition=$<$ *none* $>$
   (b) for each module on the hyperedge
   - if (module in the block) /* non-terminal */
     push the module onto a temporary stack
     continue loop (b)
   - otherwise /* terminal */
     closestPartition = { index of the subregion closest to the terminal location or $<$ *both* $>$ for equidistant subregions
   - if (closestPartition==$<$ *both* $>$) continue the loop in (b)
   - otherwise
     – if (termPartition=0)
       termPartition =closestPartition
       continue loop (b) /* skip terminal */
     – else if (termPartition$\neq$closestPartition)
       /* inessential hyperedge, ignored */
       clear stack
       break loop (b)

   (c) if (size(stack) > 1) add hyperedge connecting the modules on the stack and, if terminalPartition$\neq 0$, the respective terminal

III. Allocate block area to partition capacities in proportion to legal module locations contained in each subregion.
Assign partitioning balance tolerance on the basis of vertical/horizontal cut direction, block size and module sizes.

Figure 2: Pseudocode for splitting a block during top-down placement.

The results in Table 5 show that the best choice of thresholds yield total wirelength reductions of up to 10%, while simultaneously reducing runtime by as much as 50Overall, we believe that invoking end-case optimal bipartitioners for instance sizes of around 30-35 or less, and end-case optimal placers for instance sizes of around 7 or less, leads to good results.

4.2 Conclusions

We have shown the effectiveness of optimal partitioning and placement codes for end-case processing in top-down standard-cell placement. Our most effective implementations use branch-and-bound, with speedups due to stack-based implementation and other exploitation of the nature of the application (e.g., net cut objective, bipartitioning context, etc.). Experimental data show a surprising level of cutsize suboptimality for traditional FM partitioners, as well as a surprisingly large threshold below which branch-and-bound is faster that a single FM start. Our ongoing research explores a number of extensions of the present work, including more efficient implementations, use of multi-way optimal partitioners, and alternative partitioning and placement objectives.

References

[1] C. J. Alpert, "Partitioning Benchmarks for VLSI CAD Community", Web page, `http://vlsicad.cs.ucla.edu/~cheese/benchmarks.html` (see also the parent home page for partitioning codes).

[2] C. J. Alpert, T. Chan, D. J.-H. Huang, I. Markov and K. Yan, "Quadratic Placement Revisited", *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 752-757.

[3] C. J. Alpert, J.-H. Huang and A. B. Kahng,"Multilevel Circuit Partitioning", *ACM/IEEE Design Automation Conference*, pp. 530-533.

[4] C. J. Alpert and A. B. Kahng, "Recent Directions in Netlist Partitioning: A Survey", *Integration*, 19(1995) 1-81.

[5] J. A. Davis, V. K. De and J. D. Meindl, "A Stochastic Wire-Length Distribution for Gigascale Integration (GSI) - Part I: Derivation and Validation", *IEEE Transactions on Electron Devices*, 45(3) (1998), pp. 580-589.

[6] A. E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits", *IEEE Transactions on Computer-Aided Design* 4(1) (1985), pp. 92-98

[7] S. Dutt and W. Deng, "VLSI Circuit Partitioning by Cluster-Removal Using Iterative Improvement Techniques", *Proc. IEEE International Conference on Computer-Aided Design*, 1996, pp. 194-200

[8] S. Dutt and H. Theny, "Partitioning Around Roadblocks: Tackling Constraints With Intermediate Relaxations", *Proc. IEEE International Conference on Computer-Aided Design*, 1997, pp. 350-355.

[9] C. M. Fiduccia and R. M. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions", *Proc. ACM/IEEE Design Automation Conference*, 1982, pp. 175-181.

[10] L. Hagen, J. H. Huang and A. B. Kahng, "Quantified Suboptimality of VLSI Layout Heuristics", *Proc. ACM/IEEE Design Automation Conference*, 1995, pp. 216-221.

[11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design", *Proc. ACM/IEEE Design Automation Conference*, 1997, pp. 526-529.

[12] , B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell System Tech. Journal* 49 (1970), pp. 291-307.

Small Partitioner	Small Placer	Test Case 1 WL	Test Case 1 CPU	Test Case 2 WL	Test Case 2 CPU	Test Case 3 WL	Test Case 3 CPU	Test Case 4 WL	Test Case 4 CPU	Test Case 5 WL	Test Case 5 CPU
0	3	6.890	64	5.488	203	3.794	246	3.852	248	7.132	459
0	4	6.816	57	5.366	178	3.748	204	3.839	208	7.091	399
0	5	6.746	48	5.432	159	3.757	186	3.824	188	7.022	359
0	6	6.734	40	5.430	146	3.702	172	3.800	175	6.945	329
0	7	6.792	38	5.365	143	3.684	166	3.782	170	6.950	323
0	8	6.651	40	5.287	164	3.687	187	3.760	190	6.908	365
10	3	6.734	35	5.360	135	3.707	154	3.786	159	6.972	306
10	4	6.650	34	5.307	130	3.705	146	3.767	151	6.910	294
10	5	6.657	33	5.255	125	3.703	143	3.774	148	6.976	287
10	6	6.729	31	5.253	124	3.680	143	3.751	147	6.887	282
10	7	6.599	34	5.209	130	3.651	148	3.748	150	6.876	290
10	8	6.699	45	5.258	154	3.659	182	3.739	180	6.907	348
20	3	6.546	30	5.259	114	3.654	132	3.738	139	6.929	272
20	4	6.555	28	5.292	110	3.579	125	3.745	132	6.778	256
20	5	6.519	24	5.209	106	3.595	121	3.736	129	6.783	248
20	6	6.542	27	5.206	105	3.602	120	3.708	128	6.761	245
20	7	6.498	26	5.130	109	3.612	125	3.717	132	6.668	254
20	8	6.419	33	5.189	135	3.541	159	3.702	158	6.794	309
25	3	6.524	26	5.232	111	3.604	129	3.710	135	6.799	265
25	4	6.479	24	5.198	106	3.512	121	3.689	129	6.728	249
25	5	6.409	22	5.107	102	3.554	118	3.705	126	6.680	241
25	6	6.514	22	5.143	100	3.565	117	3.689	125	6.690	240
25	7	6.448	24	5.114	107	3.521	121	3.665	128	6.704	249
25	8	6.457	32	5.100	131	3.510	159	3.675	159	6.671	304
30	3	6.392	24	5.132	113	3.497	129	3.686	136	6.629	264
30	4	6.455	22	5.154	105	3.504	121	3.656	129	6.701	249
30	5	6.369	22	5.146	103	3.487	118	3.648	127	6.587	242
30	6	6.376	22	5.152	101	3.495	117	3.667	126	6.590	239
30	7	6.355	24	5.153	107	3.478	124	3.648	130	6.606	254
30	8	6.343	33	5.127	132	3.440	162	3.616	159	6.538	311
35	3	6.380	26	5.198	114	3.504	133	3.660	143	6.638	279
35	4	6.356	24	5.112	108	3.419	124	3.649	138	6.599	268
35	5	6.383	23	5.131	106	3.436	120	3.632	131	6.634	260
35	6	6.296	22	5.059	112	3.451	121	3.623	132	6.535	250
35	7	6.320	26	5.113	112	3.395	128	3.619	137	6.532	284
35	8	6.337	33	5.040	136	3.395	167	3.607	164	6.457	317
40	3	6.273	32	5.214	154	3.420	150	3.613	190	6.533	333
40	4	6.287	30	5.112	121	3.422	140	3.619	175	6.471	328
40	5	6.306	27	5.085	117	3.388	138	3.604	174	6.485	300
40	6	6.304	29	5.043	128	3.406	152	3.620	168	6.440	316
40	7	6.262	31	5.071	131	3.359	183	3.585	280	6.449	299
40	8	6.252	38	4.984	158	3.346	175	3.569	200	6.445	389

Table 5: Average wirelength and CPU for placements generated with various small tools thresholds. CPU time was measure on a 200Mhz Sun Sparc Ultra10.

[13] J. Kleinhans, G. Sigl, F. Johannes and K. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization", *IEEE Trans. on Computer Aided Design* 10(3) (1991), pp. 356-365.

[14] R. H. J. M. Otten, "Global Wires Harmful?", *Proc. ACM/IEEE Intl. Symp. on Physical Design*, 1998, pp. 104-109.

[15] R. H. J. M. Otten and R. K. Brayton, "Planning for Performance", *Proc. ACM/IEEE Design Automation Conference*, 1998, pp. 122-127.

[16] R. Preis and R. Diekmann, *The PARTY Partitioning-Library User Guide, Version 1.1*, University of Paderborn, September 1996.

[17] H. D. Simon and S.-H. Teng, "How Good is Recursive Bisection?", *SIAM J. Scientific Computing* 18(5) (1997), pp. 1436-1445.

[18] D. Stroobandt, "Improving Donath's Technique for Estimating the Average Interconnection Length in Computer logic", *ELIS technical report*, Royal University of Ghent, June 1996.

[19] L. Trotter, "PERM (Algorithm 115)", *Communications of the ACM* 5 (1962).

[20] R. S. Tsay and E. Kuh, "A Unified Approach to Partitioning and Placement", *IEEE Trans. on Circuits and Systems*, 38(5) (1991), pp. 521-633.

Appendix: Branch-and-Bound Pseudocodes

The input and global variables for branch-and-bound are shown in Figure 3.

Branch-and-Bound for Balanced Bipartitioning : Input and Global Variables		
Input	areaMax[0..1] upperBound hypergraph	upper bounds for partition area search for cheaper solutions node weights, #nodes, #edges
Global variables and initialization	nodeStack $=< \mathit{empty} >$ cutStack$=< \mathit{empty} >$ netStacks[0..numEdges]=$\{0\}$ areaStacks[0..1]$=< \mathit{empty} >$ nodeIdx=0 bestPartSolution$=< \mathit{invalid} >$ bestCutFound=upperBound foundLegalSolution=$false$	node-to-partition assignments "cut so far" stacks of net states "area so far" in partitions #nodes already assigned

Figure 3: Input and global variables for branch-and-bound bipartitioning. A nontrivial `upperBound` implies a known legal solution of given cost. Each `netStack` contains net states, which can represent a net with no nodes assigned to partitions, a net with nodes assigned to one partition, or a cut net.

The actual branch-and-bound algorithm is detailed in Figure 4. We note that the pseudocode shown does not work with fixed terminals, does not do anything reasonable if there are no legal solutions, and works with exactly two partitions. (Extensions to address such limitations are obvious.) We also note that efficiency requirements entail a monolithic implementation without any function calls in the critical section – in particular, we use no recursion in our implementation. However, to simplify the exposition of our algorithm, we present equivalent pseudocode that uses recursion. In a recursion-free implementation our global variables will be local variables of the monolithic function. This is why our recursion-based description is not the simplest: the recursive function has minimum local variables and does not return a value.

Balanced Bisection with Branch-and-Bound : Algorithm

```
1  assignNextNode(toPart)
2  // Assigns node with nodeIdx to a given partition
3  // in addition to previously assigned nodes with indices 0..nodeIdx.
5  {
6    if (idx<numNodes) // the solution is partial, need to branch or bound
7    {
8     weight=hypergraph.getNodeWeight(idx)
9      if ( areaStack[toPart]+weight>areaMax[toPart] ) goto bound
10     cutIncrease=0
11     for each net (netIdx) incident to curr node (idx)
12     {
13       if (netStack[netIdx].top() == 1-toPart)
14         {
15           cutIncrease = curIncrease + 1
16           netStacks[netIdx].push(< both >)
17         }
18       else if (the net does not stradde any partitions)
19         netStacks[netIdx].push(toPart)
20     }
21
22     if ( cutStack.top()+cutIncrease ≥ bestCutFound )
23     { // undo the net stacks
25       for each net (netIdx) incident to curr node (idx)
26        netStacks[netIdx].pop()
27       goto bound
28     }
30 branch: nodeStack.push(toPartition)
31     idx = idx + 1
32     areaStack[toPart].push(areaStack[toPart].top()+weight)
33     areaStack[1-toPart].push(areaStack[1-toPart].top())
34     cutStack.push(cutStack.top()+cutIncrease)
35     assignNextNode(0)
36     assignNextNode(1)
37 bound: nodeStack.pop()
38     idx = idx - 1
39     areaStack[0].pop()
40     areaStack[1].pop()
41     cutStack.pop()
42     return
43   }
44   else // have complete solution with cut < bestCutSeen
45   {
46     bestCutFound=cutStack.top()
47     copy complete solution from nodeStack to bestPartSolution
48     foundLegalSolution=true
49   }
50 }
```

Figure 4: Branch-and-bound algorithm for balanced bipartitioning is produced from a lexicographic enumeration of partitioning solutions by adding code for *bounding* in lines 9, 22-27 (shown in bold). The recursive implementation is not necessary and is used here for clarity.

Single Row Placement Branch-and-Bound Input and Data Structures		
Input	cellWidth[0..N]	width of each cell
	pinOffsets[cellId][netId]	pin-offsets (if connected) for each cell-pin pair
	terminalBoxes[netId]	bounding box of each net's terminals
	RowBox	bounding box of the row
Data Structures	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 = Infinite	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.deque()) // 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.enque(s.pop()) // remove the right-most cell
23        idx++
24        c[idx]–
25      }
26    }
27    idx–
28 }
```

Figure 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).