OPTIMAL PARTITIONERS AND END-CASE 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 study alternatives to FM-based partitioning in the context of end-case processing for top-down standard-cell placement. The primary motivation is that small partitioning instances frequently contain multiple cells larger than the prescribed partitioning tolerance (balance constraint) and cannot be moved while preserving the legality of a solution. We focus on *optimal* partitioning and placement algorithms, based on either *enumeration* or *branch-and-bound*, that are invoked for instances below prescribed size thresholds, e.g., < 10 cells for placement and < 30 cells for partitioning. Such partitioners transparently handle tight balance constraints and uneven cell sizes while typically achieving 40% smaller cuts than the best of several FM starts for instances between 10 and 35 movable nodes. On such instances, branch-and-bound codes also achieve surprising speedups, on average, over single FM starts. Enumeration-based partitioners relying on Gray codes, while easier to implement and taking less time for elementary operations, can only compete with branch-and-bound on very small instances, to which optimal placers can be applied. In the context of a top-down global placement tool, 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 *cells* 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 instance into smaller instances by subdividing the placement region, assigning cells to subregions, reformulating constraints, and cutting the netlist — such that good solutions to smaller instances (subproblems) combine into good solutions of the original problem.

*THIS WORK WAS SUPPORTED BY CADENCE DESIGN SYSTEMS, INC.

```
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 end-case
                   else
                     bipartition into smaller blocks
                     enqueue each block
```

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

In practice, the problem decomposition is accomplished by hypergraph partitioning. Each 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 [7], 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 [13, 10]; the multilevel paradigm can be applied for larger instances [3, 12]. 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), detailed placement refinement occurs.[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 cell 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 pre-

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

[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 [21] or GORDIAN [14]).

cludes 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), an alternate partitioner or placer solves the instance *optimally*. 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 instance size at which it is best to switch over from the default (FM-based) hypergraph bipartitioner to a given optimal 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 [15, 16]. 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[3] (cf. the analysis of Dutt and Theny [9]).

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., [19] [6]), 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,[4] and the pitfalls of the recursive bisection approach are well-known [18]. 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 [11]. 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.

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 optimal partitioners in a generic top-down placer implementation.

Interestingly, branch-and-bound algorithms for balanced *graph partitioning* problems have been thoroughly studied in the late 1980s and early 1990s; fast public domain implementations [17] and experimental studies [5] are available. However, few nontrivial approaches carry over to hypergraphs, and instances arising in top-down VLSI placement have not been assessed.

1.2. Contributions and Organization of Paper

In this paper, we develop new, optimal partitioners and "end-case placers" for end-case processing in top-down layout. We explore the tradeoffs between (i) exhaustive enumeration approaches based on Gray codes and (ii) branch-and-bound approaches. Section 2 describes the implementation of optimal partitioning algorithms. We compare performance of our implementations against that of LIFO- and CLIP-FM [8] for suites of small partitioning instances that arise during the top-down placement of industry standard-cell designs. The experimental data shows that our optimal partitioners enjoy runtime advantages over both LIFO- and CLIP-FM for surprisingly large instance sizes, while also yielding significantly improved solution qualities. Section 3 describes 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. OPTIMAL 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 allows cheaply maintaining cutsize during exhaustive enumeration by only updating the cut of nets incident to the moved node.
- Branch-and-bound performs depth-first traversal of a tree of *partial partitioning solutions*. A root-leaf path in this tree assigns one node at a time until complete solution is formed. With each node assignment, a lower bound on the cutsize can be updated. The lower bound converges to the actual cutsize of a complete solution when the leaf vertex is reached. The algorithm will consider completions of a partial solution only if the lower bound is smaller than the cut size of any complete solution yet seen. Without bounding, branch-and-bound would simply perform lexicographic enumeration of solutions. In the lexicographic ordering of complete partitioning solutions of N nodes, $\Theta(N)$ partition reassignments

[3] Consider a placement block with 25 cells that covers two rows and is to be bisected parallel to rows. Even, say, 5 cells each having size 2-3x larger than the average cell size (so that each such cell carries $8 - 12\%$ of the total area) leads to a situation where partition assignments of large cells are determined by the initial solution generator at every start, and are never changed by the move-based partitioner. Hence, a large number of starts may be needed to "guess" the optimal assignments of large cells with high probability.

[4] 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.

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 Code Based Optimal Partitioners

Gray code enumeration starts with all nodes in partition zero and reassigns one node at a time, always producing solutions never seen before. A Gray code for k-way partitioning of N nodes is a sequence of $2^N - 1$ numbers, each taken from the set $\{0..N-1\}$; it is interpreted as instructions to reassign respective nodes to the "next" partition modulo k.[5] The following C++ code builds such a code for `numPart`-way partitioning of `size` nodes.

```
byte* begin=_tables[size];
byte* ptr  = begin;
for(unsigned p=numPart-1; p!=0; p--)  *ptr++=0;
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 maintains partition balances and cuts for each solution it sees. A solution that satisfies balance constraints and has smallest cut seen is recorded as best. Having a lower bound for solution cost can result in a speedup (e.g., the partitioner will always stop as soon as it finds a solution of cost zero).

2.2. Branch-and-Bound Based Optimal Partitioners

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. 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. Figure 2 describes input and variables used in branch-and-bound partitioning and their initialization.

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.

2.3. Comparison of Partitioning Algorithms

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

[5] For example, the Gray code for bipartitionings of one item is { 0 }; { 0 1 0 } for two items; and { 0 1 0 2 0 1 0} for three.

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

Figure 2. 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.

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 1. Core cell, pad and net counts for test cases used.

Provenance of Small Instances

Our testbed consists of small hypergraph bipartitioning instances 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 1. 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 2. 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

Gray code enumeration was found to be 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[6] 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 cells, with Gray code enumeration being 1.9 times slower on instances with 10 cells. The runtime ratio (Gray code runtime divided by branch-and-bound runtime) increases by a factor of between 1.5 and 1.9 for each additional cell. Thus, below we compare only our branch-and-bound code against the LIFO FM and CLIP [8] algorithms. (While the Gray code enumeration is faster for instances of 8 cells or less, such instances are better handled by the end-case placers described in Section 3.)

[6] All CPU times reported are for a 300MHz Sun Ultra-10.

#non terms	#inst-ances	All Edges			Essential Edges		
		#edges	e-deg	n-deg	#edges	e-deg	n-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 2. Statistics of end-case instances for Test Case 3: average number of edges, edge and node degrees. The same statistics are shown for *essential edges* only, i.e., omitting edges that are guaranteed to be cut in any partitioning.

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 [8], 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 4 and 5 for Test Cases 1 and 3. We see that FM is clearly slower than branch-and-bound on all instances of 23 cells 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 ("suboptimal 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-cell partition-to-partition moves, while the second problem is more fundamental and can be rephrased as "FM simply makes wrong moves".

Reduction of block splitting to balanced hypergraph partitioning

Input: Original hypergraph with all cells placed at the centers of the placement regions of their blocks; A collection of cells in the block to be split; Placement region description for the block to be split (includes legal cell 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 cell in the block
2. For each hyperedge of the original (netlist) hypergraph incident to at least one of the cell in the block:
   (a) clear temporary stack for cells
   termPartition=$< none >$
   (b) for each cell on the hyperedge
   - if (cell in the block) /* non-terminal */
     push the cell onto a temporary stack
     continue loop (b)
   - otherwise /* terminal */
     closestPartition
     $= \begin{cases} \text{index of the subregion closest to the} \\ \text{terminal location or } < \ both \ > \text{ for} \\ \text{equidistant subregions} \end{cases}$
   - if (closestPartition==$< both >$)
     continue the loop in (b)
   - otherwise
     – if (termPartition=0)
       termPartition =closestPartition
       continue loop (b) /* skip terminal */
     – else if (termPartition≠closestPartition)
       /* inessential hyperedge, ignored */
       clear stack
       break loop (b)

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

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

Figure 3. Splitting a block in top-down placement.

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* cell area and either 2% or 10% of the total cell 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 cell 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. OPTIMAL 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, wirelength can be directly optimized for blocks with few nodes. In this section, we describe *optimal placers* that operate on single-row end-case instances given by:[7]

- A hypergraph with all nodes (cells) having *widths*. All cell heights are assumed equal to the row height.
- Every hyperedge has a bounding box of terminal pin locations that are incident to the respective net and fixed.
- 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.[8]

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 based approaches. Implementations based on enumeration are not competitive in our experience, 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 this lower bound is smaller than the cost of the best complete solution yet seen.

One difficulty in applying branch-and-bound to end-case placement is varying cell widths. Cells are packed with a fixed-size space between neighbors, with whitespace 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 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 4.

Single Row Placement Branch-and-Bound Input and Global Variables		
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
Variables	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.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 rightmost cell
23        idx++
24        c[idx] - -
25      }
26    }
27    idx - -
28  }
```

Figure 4. 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).

4. END-CASE PROCESSING IN GLOBAL PLACEMENT

Recall from Figure 1 that top-down placement reduces to (i) splitting blocks, and (ii) solving end-cases. We first describe our splitting algorithm because it significantly affects end-case instances. While blocks are responsible for the nets incident to their cells, our implementation does not explicitly transcribe nets from a block to its sub-blocks. Incident nets are deduced from the original netlist. Each external cell adjacent to a cell in the block is fixed at the center of its block. Thus, splitting a block reduces to balanced hypergraph partitioning with fixed terminals, as detailed in Figure 3. In particular, the possibly numerous terminals of a block are collapsed into at most two terminals in the hypergraph that is produced. Nets incident to fixed terminals in both partitions (*inessential nets*) will necessarily be cut and are therefore removed from consideration. Our implementation chooses a horizontal cutline to split a block with M cells if the block contains $M/15$ or more rows; otherwise, the choice of cut is due to the aspect ratio of the block. The blocks are split into sub-blocks as evenly as possible, so that blocks with less than 15 cells will have one row, simplifying end-case analysis.

To assess the impact of optimal partitioners and placers on top-down global placement, we have run our implementation on five industry test cases (see Table 1). We vary the two thresholds: (i) below which branch-and-bound partitioning is invoked, from 0 to 40, and (ii) below which the end-case placer is called, from 3 to 8. 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.

The best choice of thresholds in Table 3 yields total wirelength reductions of 10% while simultaneously reducing runtime by as much as 50%. Overall, invoking end-case optimal bipartitioners for instance sizes of 30-35 or less and end-case optimal placers for instance sizes of 7 or less leads to good results.

5. CONCLUSIONS

In the circuits we studied, roughly 10% of the nets in small partitioning instances do not affect partitioning solutions and can be re-

[7] End-cases have only one row because our top-down placer (see Section 4.) preferentially splits small multi-row blocks between rows.

[8] For unfortunately short subrows that cannot accommodate all cells without overlaps, our end-case placer minimizes the wirelength subject to minimum overlap.

Threshold		Test Case 1		Test Case 2		Test Case 3		Test Case 4		Test Case 5	
Part	Plac	WL	t	WL	t	WL	t	WL	t	WL	t
0	3	6.89	64	5.48	203	3.79	246	3.85	248	7.13	459
0	4	6.81	57	5.36	178	3.74	204	3.83	208	7.09	399
0	5	6.74	48	5.43	159	3.75	186	3.82	188	7.02	359
0	6	6.73	40	5.43	146	3.70	172	3.80	175	6.94	329
0	7	6.79	38	5.36	143	3.68	166	3.78	170	6.95	323
0	8	6.65	40	5.28	164	3.68	187	3.76	190	6.90	365
10	3	6.73	35	5.36	135	3.70	154	3.78	159	6.97	306
10	4	6.65	34	5.30	130	3.70	146	3.76	151	6.91	294
10	5	6.65	33	5.25	125	3.70	143	3.77	148	6.97	287
10	6	6.72	31	5.25	124	3.68	143	3.75	147	6.88	282
10	7	6.59	34	5.20	130	3.65	148	3.74	150	6.87	290
10	8	6.69	45	5.25	154	3.65	182	3.73	180	6.90	348
20	3	6.54	30	5.25	114	3.65	132	3.73	139	6.92	272
20	4	6.55	28	5.29	110	3.57	125	3.74	132	6.77	256
20	5	6.51	24	5.20	106	3.59	121	3.73	129	6.78	248
20	6	6.54	27	5.20	105	3.60	120	3.70	128	6.76	245
20	7	6.49	26	5.13	109	3.61	125	3.71	132	6.66	254
20	8	6.41	33	5.18	135	3.54	159	3.70	158	6.79	309
25	3	6.52	26	5.23	111	3.60	129	3.710	135	6.79	265
25	4	6.47	24	5.19	106	3.51	121	3.68	129	6.72	249
25	5	6.40	22	5.10	102	3.55	118	3.70	126	6.68	241
25	6	6.51	22	5.14	100	3.56	117	3.68	125	6.69	240
25	7	6.44	24	5.11	107	3.52	121	3.66	128	6.70	249
25	8	6.45	32	5.10	131	3.51	159	3.67	159	6.67	304
30	3	6.39	24	5.13	113	3.49	129	3.68	136	6.62	264
30	4	6.45	22	5.15	105	3.50	121	3.65	129	6.70	249
30	5	6.36	22	5.14	103	3.48	118	3.64	127	6.58	242
30	6	6.37	22	5.15	101	3.49	117	3.66	126	6.59	239
30	7	6.35	24	5.15	107	3.47	124	3.64	130	6.60	254
30	8	6.34	33	5.12	132	3.44	162	3.61	159	6.53	311
35	3	6.38	26	5.19	114	3.50	133	3.66	143	6.63	279
35	4	6.35	24	5.11	108	3.41	124	3.64	138	6.59	268
35	5	6.38	23	5.13	106	3.43	120	3.63	131	6.63	260
35	6	6.29	22	5.05	112	3.45	121	3.62	132	6.53	250
35	7	6.32	26	5.11	112	3.39	128	3.61	137	6.53	284
35	8	6.33	33	5.04	136	3.39	167	3.60	164	6.45	317
40	3	6.27	32	5.21	154	3.42	150	3.61	190	6.53	333
40	4	6.28	30	5.11	121	3.42	140	3.61	175	6.47	328
40	5	6.30	27	5.08	117	3.38	138	3.60	174	6.48	300
40	6	6.30	29	5.04	128	3.40	152	3.62	168	6.44	316
40	7	6.26	31	5.07	131	3.35	183	3.58	280	6.44	299
40	8	6.25	38	4.98	158	3.34	175	3.56	200	6.44	389

Table 3. Average wirelength (WL) and CPU time (t) for placements generated with various small partitioner and placer size thresholds. CPU time was measured on a 200Mhz Sun Sparc Ultra10.

moved. Our experiments also show that optimal partitioners based on branch-and-bound are easy to implement and outperform FM-based heuristics by as much as 40% on problem instances of up to 30 nodes; they are also faster than a single start of FM when there are fewer than 25 nodes.

Given that FM always stops after the first non-improving pass, our findings are rather surprising. We believe that even with algorithm modifications to find better solutions (e.g., [9]), FM-based algorithms are not likely to compete with branch-and-bound on small instances. At the same time, the huge suboptimality of FM solutions suggests that other move-based partitioning algorithms (e.g., simulated annealing) for which maintaining legality of the current solution is important may perform poorly on small instances with non-uniform cell sizes.[9]

While our experiments have been limited to available benchmark circuits, the overall superiority of optimal end-case processors should carry over to larger circuits, where the number of small partitioning instances will increase with circuit size, while the same relative improvement in quality is likely. More efficient branch-and-bound codes are undoubtedly possible, and their study is the subject of ongoing research. Other important questions include the use of multi-way partitioners as well as alternative partitioning and placement objectives.

[9] Many popular move-based partitioning algorithms were originally proposed for *uniform* cell sizes and have not been extensively studied otherwise. While they trivially apply to non-uniformly sized cells as well, their performance may deteriorate for reasons not previously considered.

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. Clausen and J. L. Träff, "Do Inherently Sequential Branch-and-Bound Algorithms Exist?", *Parallel Processing Letters* 4(1-2) (1994), pp. 3-13.

[6] 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.

[7] 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

[8] 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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

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

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

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

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

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

[21] 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.

TEST CASE 1 / LIFO FM											TEST CASE 3 / LIFO FM										
size	#good	#SOpt	1 start		2 starts		3 starts		100 starts		size	#good	#SOpt	1 start		2 starts		3 start		100 starts	
	inst	inst	time	cut	time	cut	time	cut	time	cut		inst	inst	time	cut	time	cut	time	cut	time	cut
10	24(**20**)	7	23.23	2.035	46.47	1.796	69.70	1.670	2323.4	1.175	10	160(**134**)	32	20.73	1.976	41.46	1.700	62.19	1.564	2073.1	1.080
11	37(**30**)	6	16.63	2.018	33.26	1.730	49.89	1.591	1663.1	1.064	11	145(**130**)	25	18.84	2.112	37.69	1.803	56.54	1.651	1884.7	1.069
12	31(**26**)	1	17.24	2.291	34.49	1.961	51.73	1.799	1724.5	1.020	12	94(**83**)	8	17.02	1.948	34.05	1.671	51.08	1.537	1702.7	1.029
13	22(**19**)	2	22.65	2.199	45.30	1.867	67.95	1.711	2265.0	1.029	13	85(**81**)	10	16.10	2.054	32.21	1.757	48.32	1.609	1610.8	1.030
14	22(**21**)	7	14.29	2.037	28.58	1.766	42.87	1.639	1429.2	1.069	14	58(**55**)	11	11.14	1.892	22.29	1.623	33.44	1.496	1114.9	1.042
15	20(**20**)	4	11.46	2.001	22.92	1.734	34.38	1.617	1146.0	1.056	15	78(**76**)	24	10.13	1.840	20.27	1.603	30.41	1.496	1013.7	1.059
16	9(**9**)	4	10.13	1.690	20.26	1.493	30.40	1.404	1013.3	1.095	16	65(**62**)	20	6.796	1.846	13.59	1.634	20.38	1.530	679.6	1.053
17	12(**11**)	4	7.066	1.887	14.13	1.677	21.19	1.579	706.6	1.077	17	68(**68**)	32	5.422	1.933	10.84	1.713	16.26	1.611	542.2	1.118
18	6(**6**)	5	8.281	1.915	16.56	1.722	24.84	1.639	828.0	1.256	18	40(**40**)	25	4.430	1.907	8.860	1.717	13.29	1.628	443.0	1.149
19	8(**8**)	5	9.344	2.315	18.68	2.007	28.03	1.857	934.3	1.173	19	47(**46**)	38	3.577	1.967	7.154	1.775	10.73	1.681	357.7	1.214
20	11(**11**)	9	3.562	2.340	7.124	2.088	10.68	1.959	356.2	1.275	20	42(**40**)	29	2.761	1.913	5.523	1.726	8.284	1.635	276.1	1.178
21	12(**12**)	10	3.361	2.258	6.723	2.027	10.08	1.916	336.1	1.257	21	44(**44**)	39	2.191	2.000	4.382	1.806	6.573	1.711	219.1	1.228
22	10(**10**)	9	3.831	2.099	7.662	1.904	11.49	1.800	383.1	1.242	22	27(**27**)	22	1.429	2.001	2.857	1.810	4.286	1.721	142.8	1.217
23	7(**7**)	7	1.312	2.166	2.624	1.979	3.936	1.884	131.2	1.371	**23**	37(**37**)	36	**1.134**	1.969	2.268	1.806	3.402	1.721	113.4	1.275
24	8(**8**)	8	1.187	2.154	2.373	1.968	3.560	1.877	118.6	1.394	24	30(**30**)	27	0.871	2.088	1.743	1.896	2.614	1.805	87.14	1.294
25	7(**7**)	7	**1.325**	2.342	2.651	2.114	3.976	2.003	132.5	1.403	25	32(**32**)	32	0.826	2.159	1.652	1.993	2.478	1.905	82.60	1.415
26	11(**11**)	11	0.703	2.473	1.405	2.266	2.108	2.158	70.25	1.503	26	38(**38**)	38	0.512	2.368	**1.023**	2.171	1.535	2.072	51.16	1.512
27	8(**8**)	8	0.662	2.405	1.324	2.183	1.986	2.083	66.18	1.482	27	34(**34**)	31	0.495	2.198	0.990	2.010	1.484	1.913	49.47	1.354
28	10(**10**)	10	0.418	2.522	0.835	2.286	1.253	2.168	41.77	1.403	28	31(**31**)	31	0.357	2.227	0.713	2.054	**1.070**	1.963	35.67	1.468
29	9(**9**)	9	0.746	2.316	1.492	2.118	2.238	2.019	74.59	1.434	29	21(**21**)	19	0.261	2.201	0.523	2.031	0.784	1.939	26.13	1.434
30	2(**2**)	2	1.026	3.094	2.052	2.803	3.078	2.654	102.58	1.789	30	25(**25**)	24	0.151	1.973	0.302	1.834	0.453	1.765	15.11	1.390
31	7(**7**)	4	0.596	1.958	1.192	1.811	1.788	1.743	59.59	1.474	31	12(**12**)	10	0.251	2.000	0.502	1.868	0.753	1.805	25.10	1.465
32	4(**4**)	2	0.675	2.196	**1.351**	1.930	**2.026**	1.800	67.53	1.273	32	13(**13**)	9	0.261	1.698	0.522	1.595	0.783	1.550	26.08	1.287
33	1(**1**)	1	0.213	3.046	0.427	2.801	0.640	2.700	21.33	2.143	33	9(**9**)	7	0.106	1.903	0.211	1.782	0.317	1.720	10.56	1.397
34	3(**3**)	3	0.142	2.453	0.285	2.258	0.427	2.151	14.23	1.641	34	13(**13**)	13	0.078	2.773	0.155	2.562	0.233	2.447	7.759	1.816
35	2(**2**)	2	0.007	2.062	0.014	1.932	0.021	1.854	0.707	1.401	35	9(**9**)	9	0.052	2.326	0.104	2.183	0.157	2.111	5.218	1.678
TEST CASE 1 / CLIP FM											TEST CASE 3 / CLIP FM										
10	24(**20**)	9	24.08	1.938	48.166	1.710	72.248	1.600	2408.2	1.180	10	160(**124**)	27	24.23	1.971	48.47	1.688	72.71	1.552	2423.8	1.070
11	37(**31**)	7	22.60	1.992	45.209	1.692	67.814	1.552	2260.4	1.057	11	145(**120**)	20	21.66	2.129	43.33	1.819	65.00	1.666	2166.6	1.056
12	31(**26**)	3	18.94	2.134	37.899	1.839	56.848	1.700	1894.9	1.040	12	94(**86**)	9	17.96	1.985	35.93	1.698	53.90	1.563	1796.8	1.035
13	22(**17**)	6	18.00	2.248	36.005	1.910	54.007	1.762	1800.2	1.105	13	85(**77**)	7	15.76	2.005	31.52	1.712	47.29	1.572	1576.3	1.023
14	22(**19**)	4	15.05	2.085	30.113	1.802	45.169	1.667	1505.6	1.046	14	58(**55**)	9	10.47	1.867	20.95	1.601	31.43	1.473	1047.9	1.036
15	20(**20**)	7	14.95	2.018	29.899	1.749	44.849	1.628	1494.9	1.091	15	78(**77**)	24	10.68	1.867	21.37	1.625	32.05	1.508	1068.6	1.068
16	9(**9**)	2	10.09	1.709	20.184	1.507	30.276	1.420	1009.2	1.026	16	65(**65**)	26	7.488	1.890	14.97	1.670	22.46	1.564	748.7	1.099
17	12(**12**)	4	6.797	1.851	13.595	1.648	20.392	1.549	679.7	1.072	17	68(**68**)	35	5.959	1.945	11.91	1.728	17.87	1.623	595.8	1.133
18	6(**5**)	4	7.477	1.972	14.953	1.788	22.430	1.702	747.6	1.200	18	40(**40**)	26	3.926	1.908	7.851	1.720	11.77	1.623	392.5	1.157
19	8(**8**)	5	8.437	2.335	16.875	2.014	25.312	1.862	843.7	1.218	19	47(**47**)	36	3.481	1.965	6.962	1.774	10.44	1.678	348.1	1.198
20	11(**9**)	7	3.683	2.385	7.366	2.130	11.049	1.992	368.3	1.227	20	42(**42**)	29	3.150	1.922	6.301	1.736	9.451	1.645	315.0	1.177
21	12(**12**)	11	3.882	2.270	7.764	2.038	11.646	1.922	388.1	1.269	21	44(**43**)	35	2.276	1.989	4.552	1.806	6.827	1.714	227.5	1.213
22	10(**10**)	9	2.717	2.117	5.433	1.920	8.150	1.827	271.6	1.285	22	27(**27**)	21	1.422	1.999	2.843	1.817	4.265	1.720	142.1	1.245
23	7(**7**)	7	1.316	2.158	2.633	1.964	3.949	1.867	131.6	1.354	23	37(**37**)	34	**1.186**	1.979	2.372	1.813	3.558	1.733	118.5	1.296
24	8(**8**)	7	1.334	2.126	2.668	1.941	4.001	1.851	133.3	1.321	24	30(**30**)	29	0.923	2.100	1.846	1.912	2.769	1.818	92.29	1.300
25	7(**7**)	6	**1.387**	2.359	2.775	2.114	4.162	1.997	138.7	1.283	25	32(**32**)	32	0.779	2.151	1.559	1.974	2.338	1.885	77.92	1.398
26	11(**11**)	11	0.618	2.461	1.236	2.253	1.853	2.149	61.77	1.497	26	38(**38**)	37	0.519	2.380	1.037	2.185	1.556	2.086	51.86	1.541
27	8(**8**)	7	0.544	2.406	1.089	2.171	1.633	2.062	54.44	1.370	27	34(**34**)	33	0.585	2.199	**1.169**	2.008	1.754	1.912	58.45	1.374
28	10(**10**)	10	0.389	2.527	0.778	2.305	1.167	2.199	38.91	1.671	28	31(**31**)	31	0.361	2.219	0.723	2.038	**1.084**	1.947	36.13	1.421
29	9(**9**)	9	0.792	2.320	1.583	2.116	2.375	2.018	79.15	1.394	29	21(**21**)	20	0.242	2.183	0.485	2.011	0.727	1.925	24.24	1.439
30	2(**2**)	2	1.772	3.049	3.543	2.807	5.315	2.695	177.1	1.891	30	25(**25**)	24	0.155	1.988	0.311	1.849	0.466	1.781	15.53	1.369
31	7(**7**)	4	0.624	1.930	1.247	1.788	1.871	1.734	62.36	1.393	31	12(**12**)	10	0.248	2.002	0.496	1.865	0.744	1.799	24.80	1.393
32	4(**4**)	2	0.921	2.206	**1.842**	1.982	**2.763**	1.878	92.09	1.185	32	13(**13**)	9	0.289	1.691	0.578	1.593	0.867	1.554	28.88	1.305
33	1(**1**)	1	0.217	3.021	0.433	2.778	0.650	2.672	21.66	2.000	33	9(**9**)	7	0.104	1.913	0.209	1.791	0.313	1.731	10.43	1.374
34	3(**3**)	3	0.120	2.464	0.241	2.280	0.361	2.179	12.02	1.689	34	13(**13**)	13	0.080	2.747	0.161	2.540	0.241	2.427	8.049	1.816
35	2(**2**)	2	0.007	2.074	0.015	1.932	0.022	1.866	0.731	1.477	35	9(**9**)	9	0.052	2.327	0.105	2.178	0.157	2.103	5.230	1.613

Table 4. Comparison of LIFO FM and CLIP FM against Branch-and-Bound for Test Case 1, 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. "Good" instances have non-zero optimal cut and are sufficiently difficult for branch-and-bound so that CPU time can be measured (this can only make FM look better). "Suboptimal" (SOpt) instances are those on which FM failed to find the optimum solution in 100 random starts. Ratios greater than 1.0 indicate FM losses. Transition points for run time are shown in bold.

Table 5. Comparison of LIFO FM and CLIP FM against Branch-and-Bound for Test Case 3, 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. "Good" instances have non-zero optimal cut and are sufficiently difficult for branch-and-bound so that CPU time can be measured (this can only make FM look better). "Suboptimal" (SOpt) instances are those on which FM failed to find the optimum solution in 100 random starts. Ratios greater than 1.0 indicate FM losses. Transition points for runtime are shown in bold.