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

Andrew E. Caldwell, Andrew B. Kahng, and Igor L. Markov, *Member, IEEE*

***Abstract*—We study alternatives to classic Fiduccia–Mattheyses (FM)-based partitioning algorithms in the context of end-case processing for top-down standard-cell placement. While the divide step in the top-down divide and conquer is usually performed heuristically, we observe that optimal solutions can be found for many sufficiently *small partitioning instances*. Our main motivation is that small partitioning instances frequently contain multiple cells that are larger than the prescribed partitioning tolerance, and that cannot be moved iteratively while preserving the legality of a solution. To sample the suboptimality of FM-based partitioning algorithms, 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 best of several FM starts for instances between ten and 50 movable nodes and average degree 2–3. Our branch-and-bound codes incorporate various efficiency improvements, using results for hypergraphs from [14] and a graph-specific algorithm from [22]. We achieve considerable speed-ups over single FM starts on such instances *on average*. 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, where optimal placers achieve reasonable performance as well. 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.**

Our results show that run-time versus quality tradeoffs may be different for small problem instances than for common large benchmarks, resulting in different comparisons of optimization algorithms. We therefore suggest that alternative algorithms be considered and, as an example, present detailed comparisons with the flow-based balanced partitioner heuristic [19].

***Index Terms*—Algorithms, design automation, graph theory, Gray codes, integrated circuit layout, very-large-scale integration.**

I. Introduction

In THE placement phase of physical design for standard-cell very large scale integrated (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 instance into smaller instances by subdividing the placement region, assigning modules to subregions, reformulating constraints, and cutting the netlist—such that good solutions to smaller instances (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 [9], and hyperedges are induced over the node set from the original netlist. The actual hypergraph partitioning is performed using Fiduccia–Mattheyses (FM)-type iterative partitioning heuristics with minimum net cut objective [16], [12]; the multilevel paradigm can be applied for larger instances [3], [15]. After a global placement solution has been found (a minimum requirement being that all cells be 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 Fig. 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-submicrometer designs is often small; hence, 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.

Manuscript received November 7, 1999. This paper was recommended by Associate Editor M. Wong.

A. E. Caldwell is with Simplex Solutions, Inc., 521 Almanor Ave., Sunnyvale, CA 94086 USA.

A. B. Kahng is with the UCLA Computer Science Department., Los Angeles, CA 90095-1596 USA.

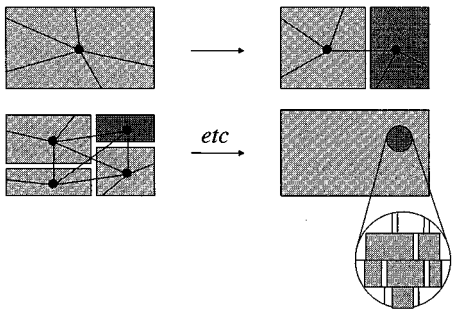
I. L. Markov is with the EECS Department, University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122 USA.

Publisher Item Identifier S 0278-0070(00)10287-8.

[1] A *block* conceptually corresponds to 1) a placement region with allowed locations, 2) a collection of modules to be placed in this region, 3) all nets incident to the modules, and 4) locations of all modules beyond the given region that are adjacent to some modules 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 [26] or GORDIAN [17]).

0278–0070/00$10.00 © 2000 IEEE




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

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

As shown in Fig. 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 instance size at which it is best to switch over from the default (FM-based) hypergraph bipartitioner to a given end-case algorithm.

A. Motivations for Optimal End-Case Processing

With each new deep-submicrometer process generation, there is a wider range of cell sizes in cell libraries. For example, an 80× range of inverter strengths is not uncommon today, and the number of complex gates in the library also increases. Overall, the wider range of cell sizes is due to the wider range of interconnect layer RC parameters, and to new methodologies for achieving performance convergence via sizing-based optimizations [20], [21]. 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: 1) 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 2) 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 [11]).

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., [24] [8]), 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 [23]. 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 [13]. 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: 1) to assess the cutsize suboptimality of traditional FM-based approaches for small partitioning instances arising in top-down placement, 2) assess the runtime penalty that can also be incurred with traditional FM-based approaches, and 3) determine the overall effect of new "end-case placers and partitioners" in a generic top-down placer implementation.

Interestingly, branch-and-bound algorithms for balanced *graph partitioning* problems have been well-studied in the late 1980s and early 1990s: fast public domain implementations [22] and experimental studies [7] are available. However, few nontrivial approaches carry over to hypergraphs, and instances arising in top-down placement of VLSI circuits have not been assessed.

B. Contributions and Organization of Paper

In this paper, we improve the top-down placement heuristics with optimal partitioners for small instances and optimal

[3]For a simple-minded example, consider a placement block with 25 nodes that covers two rows and needs to be bisected parallel to rows. If cells were uniformly sized, each would take 4% of the total cell area, which is already beyond the traditional 2% tolerance commonly used to make benchmarks "tight." Having a group of, say, 5 cells that are two to three times bigger than the smallest cell, we arrive at a situation where partition assignments of large cells are determined by the initial solution generator at every start and never change over the progression of the move-based partitioner. In this case, at least 2^4 starts may be necessary 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 As, the Bs, the Cs, and the Ds, along with an A_1–C_1 edge and a B_1–D_1 edge. The cliques over the Bs and Ds have weight 2/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 As and Bs in Partition 0, and all Cs and Ds 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 zero.

end-case placers. We explore the tradeoffs between: 1) exhaustive enumeration approaches based on Gray codes and 2) branch-and-bound approaches. Section II describes the implementation of optimal partitioning algorithms. We compare performance of our implementations against that of LIFO- and CLIP-FM [10] 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 III describes the implementation of optimal linear placement algorithms. Section IV 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.

In addition to the above, we provide average runtimes in seconds and average cutsizes for our experiments. We also give an additional comparison to the flow-based balanced (FBB) partitioner [19], which appears competitive to FM and branch-and-bound in certain size ranges.

II. 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 permits easy maintenance of 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 a complete solution is obtained. 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. The algorithm will only consider extensions of a partial solution if the lower bound is smaller than the cutsize of any complete solution 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 are required on average between successive solutions. Thus, effective bounding is necessary for branch-and-bound to be faster than Gray code-based enumeration.

A. 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 numbers $0 \cdots N-1$ of length $2^N - 1$ that can be 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 speed-up, e.g., the partitioner will stop once it finds a solution of cost zero.

B. Branch-and-Bound Based Optimal Partitioners

We first describe a fairly straightforward branch-and-bound algorithm for general hypergraphs and then discuss speed-ups for cases when all hyperedges have degree two. We are able to combine the two in our implementation by decomposing the hypergraph into "proper hyperedges" and "the graph part" to maintain cuts separately.

1) Algorithm for General Hypergraphs: 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. Fig. 2 describes input and variables used in branch-and-bound partitioning and their initialization.

The algorithm operates on a "main stack" that: 1) stores partition assignments for all nodes assigned so far and 2) 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

[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 Hypergraph Bipartitioning : Input and Global Variables

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

Fig. 2. Input and global variables for branch-and-bound bipartitioning. A nontrivial `upperBound` implied 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.

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) Speed-Ups for Hyperedges of Degree Two: An improved cost computation is possible for graphs [22]. This leads to a specialized branch-and-bound that proceeds in the same way as for general hypergraphs, but can "see" higher costs given the same nodes assigned to the same partitions. The result is a more efficient bounding capability and a sizable speed-up. Motivating examples of improved cost computation are given in Fig. 3(a) and (b) where dark nodes are assigned to the left and right partitions, while the light node is not yet assigned. Note that no edges incident to the light node are cut. However, either assignment of the light node will result in the additional cut of one in (a), and at least two in (b). Therefore we say that there is an *inevitable cut* associated with the unassigned node.

To compute *inevitable cut* for a given unassigned node, we maintain counts of adjacent nodes assigned to each partition. The inevitable cut is the smaller of the two numbers.[6] The maintenance is performed when a node is assigned or unassigned, whence all adjacent unassigned nodes are traversed and their counts updated. At the same time, we can also update the overall cut estimate.

The only aspect of incidence information that we need here is, for every node, the list of nodes adjacent to it that are *further* in the node order used in branch-and-bound. In other words, there is no need to test whether a node is assigned or not.

3) Hybrid Cost Maintenance: Before running the main branch-and-bound loop that has been described earlier, we can represent the two-pin edges ("the graph part") of the original hypergraph separately from the "proper hyperedges." Whenever a node is assigned or unassigned, we perform relevant

[6]In the multiway case, the inevitable cut is computed by subtracting the largest of the adjacent node counts from the degree.

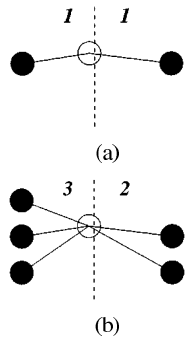


Fig. 3. Computation of *inevitable cuts*. In each figure, the dark nodes are assigned to the left and right partitions; the light node is not yet assigned. Inevitable cuts are (a) one and (b) two.

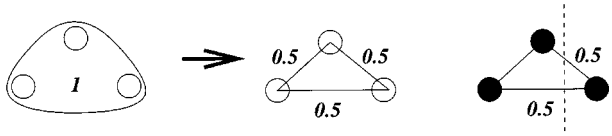


Fig. 4. Conversion of hyperedges of degree three into triangles preserves cuts.

operations with *both* representations and add the resulting lower bounds for cut. Since the two representations are disjoint and their union contains all original hyperedges, the sum is a lower bound for the cut of the original hypergraph.

In our computation, only edges of degree two can cause inevitable cuts. At the same time, it is surprisingly difficult to extend inevitable cuts to hyperedges while still achieving a lower bound for cut; cf. the review of hyperedge models given by Lengauer [18]. We apply a technique of converting all degree-3 hyperedges into triangles of three degree-2 graph edges [14], each with weight 0.5 (see Fig. 4). To see that for all assignments of the three nodes this produces the correct cut compared to the original hyperedge, we observe that there are only two different configurations. If all nodes are not in one partition (in which case the cut is 0), then exactly two edges of the triangle must be cut, which results in cut $0.5 \cdot (1+1)$.

If the original degree-3 hyperedge has nontrivial weight, then the weights of the newly created degree-2 edges are simply half the weight of the hyperedge. If all original weights are integral, one can avoid floating-point arithmetic by multiplying all weights by two, as this cannot influence the optimality of particular partitioning solutions.

C. 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.

1) 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 IV below. We have saved all instances with 10–99 (movable) *nonterminal* nodes that arise during the top-down placement of testcase 1 and testcase 3, out of the five industrial test cases described in Table I below. These small instances have fairly uniform statistical properties across designs that we have seen.

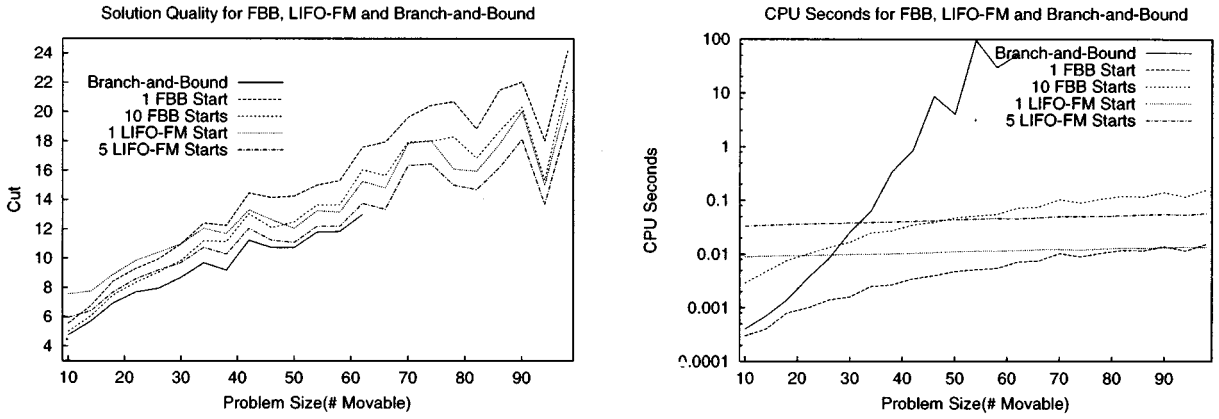


Fig. 5. Comparison of our Branch and Bound methods with one or ten starts of FBB and one or five starts of FM on problems saved from our placement tool. Data expressed as (average cut/average CPU time); time is expressed in CPU seconds on a 300-MHz Sun Ultra-10.

Typical statistics given in [6] include average hyperedge degree 2.2–2.3 and average node degree 3.2–3.6.

2) Runtime Comparisons Versus 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.[7] Our two implementations perform comparably on instances with nine modules, with Gray code enumeration being 1.9 times slower on instances with ten modules. 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 module. Thus, we have compared only our branch-and-bound code against the LIFO FM and CLIP [10] algorithms. (While the Gray code enumeration is faster for instances of eight modules or less, such instances are better handled by the end-case placers described in Section III.)

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 two, three and one hundred 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 two, three or one hundred starts of FM are used. All ratios are averaged geometrically. Finally, we repeat the entire experiment using the CLIP algorithm of Dutt and Deng [10], which is in general a stronger flat partitioner. We note that our FM implementation is at least as fast and obtains at least as good solution quality on average compared to the public-domain implementation of W. Deng that is available from C. J. Alpert's web page [1]. Our CLIP implementation compares similarly to other reported implementations.

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

Experimental results are shown in Fig. 5.[8] We see that FM is clearly slower than branch-and-bound on instances of 27 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 I, small balanced hypergraph partitioning instances are inherently difficult for FM, CLIP and any move-based partitioners because of poor reachability in the solution space caused by the balance constraint. This means that not all feasible solutions can be reached from a given solution by legal single-module partition-to-partition moves. Even if, in the strict sense, all solutions can be reached from a given solution, it may be difficult for FM to find good solutions due to "bottlenecks" in the solution space.

To discourage overconstrained instances and thus improve the reachability in the solution space, all partitioning instances produced by our placer have at least 10% tolerance.[9] Early results of our research reported at ISPD-99 [6] used partitioning instances with much tighter tolerances, whence FM performed considerably worse than on our current partitioning instances. However, even with 10%+ tolerance, branch-and-bound is preferable to FM for sufficiently small instances.

3) Pathological Cases: During our experiments we observed two types of pathologies in partitioning instances. In such instances, the failure to bound away large parts of solution space leads to enormous increase in required CPU time. Bounding can be easily defeated when the instance has a large number of optimal solutions and most vertex assignments do not change the cost of partial solution.[10] In Fig. 6(a), a mul-

[8]The ruggedness of the curves is due to the pool of partitioning instances saved. In particular, top-down placement generates fewer instances of larger size; the right side of plots is more rugged as averaging becomes less effective.

[9]Possible overlaps in placement are removed by a post-processing step after each level of top-down placement.

[10]This is especially apparent when *inevitable cuts* are used and allow computing correct cost very early. We note, however, that inevitable cuts can never slow down branch-and-bound.

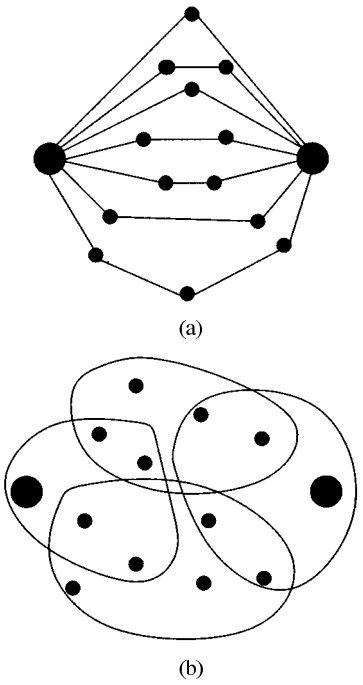


Fig. 6. Pathological cases for branch-and-bound partitioning, having large numbers of optimal solutions. Larger balls represent fixed vertices; smaller balls represent movable vertices.

titude of optimal solutions arises due to disconnected "threads" each of which has several assignments compatible with optimal solutions. High-degree hyperedges in Fig. 6(b) affect the cost function very little, and the instance is effectively very sparse. In general, any instance with numerous symmetries can be expected to cause trouble for branch-and-bound; this includes extremely sparse and extremely dense instances because they are very close to highly symmetric instances. In this context, we note that we observed more pathological dense instances [some similar to Fig. 6(a)] when higher-level heuristic partitioners were detuned to perform poorly. Another observation is that FM typically performs well on instances that have many optimal solutions and is not affected by symmetries *per se*. Therefore, it is reasonable to limit the CPU time given to branch-and-bound, and call FM if branch-and-bound times out.

III. Optimal Placement

In the top-down partitioning-based placement approach, the original placement problem (considered as a "block") is partitioned into two subproblems (subblocks) and then recursively into smaller and smaller subproblems (recall Fig. 1). Eventually, wirelength can be directly optimized for blocks with few nodes. We now describe *optimal placers* that operate on arbitrary single-row end-case instances given by:[11]

- A hypergraph with nodes (cells) having (x, y)-dimensions. All cell heights are assumed equal to the row height.
- Every hyperedge has a bounding box of fixed pin locations corresponding to the external terminals incident to that net.
- Each hyperedge-to-node connection has a *pin offset* relative to the cell origin.
- A placement region, i.e., a subrow of a certain length.[12]

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

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

One difficulty in applying branch-and-bound to end-case placement is varying cell widths. We restrict cells in the small instance to be packed with a fixed-size space between neighbors, i.e., whitespace is distributed equally between them. Replacing a cell with a cell of different width will change the location of at least one neighbor, triggering bounding box recomputations for incident nets. To simplify maintenance, the nodes are packed from left to right and 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 Fig. 7. Typical runtimes of our resulting end-case placer implementation, within the experimental context described in Section IV, are 0.042 s for eight-cell instances, 0.011 s for 7-cell instances, and 0.002 s for six-cell instances on a 300-MHz Sun Ultra-10.

IV. End-Case Processing in Global Placement

Recall from Fig. 1 that top-down placement reduces to 1) splitting blocks, and 2) solving endcases. We first describe our splitting algorithm because it significantly affects end-case instances. While blocks are responsible for the nets incident to their modules, our implementation does not explicitly transcribe nets from a block to its subblocks. Incident nets are deduced from the original netlist. Each external module adjacent to a module 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 Fig. 8. In particular, the possibly numerous terminals of a block are collapsed into at most two terminals in the resulting hypergraph. 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 modules if the block con-

[11]End-cases have only one row because our top-down placer described in Section IV preferentially splits small multirow blocks between rows.

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

Single Row Placement Branch-and-Bound
Input and Data Structures

Input	cellWidth[0..N]	width of each cell
	pinOffsets[cellId][netId]	pin-offsets for each cell-pin pair
	terminalBoxes[netId]	bounding boxes of net terminals
	RowBox	bounding box of the row
Data Struct	nodeQueue =[0....N-1]	inverse initial ordering
	nodeStack=< *empty* >	placement ordering
	counterArray=< *empty* >	loop counter array
	idx=$N-1$	index
	costSoFar= 0	cost of the current placement
	bestYetSeen = 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 right-most cell
23      idx++
24      c[idx]–
25    }
26   }
27   idx–
28 }
```

Fig. 7. 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).

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≠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≠ 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.

Fig. 8. Splitting a block in top-down placement.

tains $M/15$ or more rows, otherwise the choice of cut is due to the aspect ratio of the block. The blocks are split into subblocks as evenly as possible, therefore blocks with less than 15 cells will have one row, simplifying endcase analysis.

To assess the impact of optimal partitioners and placers on top-down global placement, we have run our implementation on five test cases (see Table I) supplied from the industry in the Cadence Design Systems LEF/DEF format. In these experiments, we varied instance size thresholds: 1) below which branch-and-bound partitioning is invoked—from 0 to 40, and 2) below which the end-case placer is called—from 3 to 8. All applications of FM consist of five 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[13] in Table II 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 seven or less leads to good results.

[13]In our experience, this choice depends on several features of the top-down placer, e.g., the optimal thresholds were higher after improvements in the heuristic partitioners applied at several previous levels. Therefore, fine-tuning is recommended in practice. We also note a less pronounced *instance*-dependency.

TABLE I
CORE CELL, PAD AND NET COUNTS FOR TEST CASES USED

Test Case	Core Cells	Pads	Nets
1	11471	662	11673
2	12146	711	10851
3	20392	185	21987
4	35549	121	44121
5	85385	177	87272

V. CONCLUSION

Our experiments show that optimal partitioners based on branch-and-bound are easy to implement and outperform FM-based heuristics by 20%–70% on problem instances of up to 40 nodes; they are also faster than a single start of FM when there are fewer than 27 nodes. However, solutions

TABLE II
AVERAGE WIRELENGTH (WL) AND TIME (t) FOR PLACEMENTS GENERATED WITH VARIOUS SMALL PARTITIONER AND PLACER SIZE THRESHOLDS. CPU TIME WAS MEASURED ON A 140-MHz SUN SPARC ULTRA1

Thresholds		Test Case 1		Test Case 2		Test Case 3		Test Case 4		Test Case 5	
Part'ers	Placers	WL	time	WL	time	WL	time	WL	time	WL	time
3	0	2.922	513	2.810	520	6.335	933	10.43	1992	2.610	5180
3	10	2.806	369	2.691	370	6.180	680	10.01	1419	2.521	3820
3	20	2.786	317	2.649	328	6.078	585	9.853	1382	2.457	3187
3	25	2.785	319	2.633	300	5.979	640	9.871	1464	2.516	3387
3	30	2.777	354	2.608	307	5.970	632	9.790	2478	2.502	3763
3	35	2.765	383	2.625	372	5.941	753	9.720	4014	2.391	4144
4	0	2.900	466	2.820	426	6.272	876	10.36	1615	2.537	4724
4	10	2.790	327	2.693	332	6.108	672	9.963	1259	2.428	3598
4	20	2.754	292	2.640	291	6.040	577	9.753	1273	2.428	3454
4	25	2.762	341	2.632	308	5.957	592	9.727	1468	2.471	3376
4	30	2.759	376	2.610	331	6.037	705	9.695	2739	2.426	3951
4	35	2.765	453	2.631	395	5.872	782	9.657	5086	2.396	3820
5	0	2.869	407	2.778	405	6.363	739	10.18	1831	2.486	4866
5	10	2.799	358	2.697	306	6.061	624	9.892	1540	2.413	4171
5	20	2.751	317	2.624	280	6.015	574	9.801	1425	2.489	3547
5	25	2.754	338	2.620	342	5.951	639	9.666	1650	2.568	3793
5	30	2.759	337	2.600	268	5.961	564	9.675	2600	2.439	3630
5	35	2.755	422	2.613	372	6.012	830	9.608	4658	2.354	4387
6	0	2.837	385	2.764	423	6.265	719	10.07	1723	2.462	3778
6	10	2.788	392	2.677	345	6.083	675	9.903	1421	2.396	3515
6	20	2.744	286	2.611	284	5.986	592	9.795	1502	2.349	3274
6	25	2.746	284	2.634	266	5.971	552	9.696	1368	2.410	3053
6	30	2.753	309	2.640	294	5.904	557	9.593	2453	2.420	3149
6	35	2.749	385	2.639	403	5.986	901	9.581	5516	2.476	4093
7	0	2.823	463	2.746	425	6.138	763	9.944	1764	2.507	4520
7	10	2.784	404	2.681	362	6.068	725	9.902	1532	2.439	4125
7	20	2.743	365	2.651	325	5.994	639	9.636	1388	2.429	3556
7	25	2.753	336	2.649	266	5.902	513	9.593	1505	2.486	3239
7	30	2.739	325	2.614	301	5.949	611	9.549	2664	2.493	3349
7	35	2.754	348	2.627	306	5.901	735	9.547	4233	2.488	4178
8	0	2.809	484	2.729	472	6.097	927	9.862	1940	2.483	4684
8	10	2.782	434	2.680	460	6.049	905	9.790	1699	2.491	4620
8	20	2.745	414	2.677	423	5.958	789	9.594	1679	2.412	4240
8	25	2.731	408	2.639	390	5.906	785	9.537	1963	2.462	3939
8	30	2.741	417	2.626	416	5.890	791	9.517	3122	2.423	4086
8	35	2.736	465	2.638	432	5.929	891	9.463	5170	2.449	4371

produced by FM for instances of 60 and more nodes are most often within 10% of optimal and this gap decreases further as instances grow. The fact that the CPU time taken by FM on those instances is orders of magnitude smaller than that taken by branch-and-bound, reconfirms FM as a leading heuristic for *medium* and *large-scale* hypergraph partitioning. However, even with algorithm modifications to find better solutions (e.g., [11]), FM-based algorithms are not likely to compete with branch-and-bound on small instances. The considerable 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 nonuniform cell sizes.[14] Alternate algorithmic approaches may be better suited to the nature of end-case instances. Toward this end we compare FM and branch-and-bound to the flow-based FBB heuristic [19] by H. Liu and M. Wong from the University of Texas at Austin, who graciously offered their implementations to us.[15]

The detailed results of those experiments are available in Fig. 9, where average cuts and run times are given that partitioners achieve on instances of sizes 10–100. One start of FBB is always faster than branch-and-bound, and is also faster than one start of FM for instances with up to 90 nodes. However, the solution quality of the ten-start FBB is suboptimal by more

[14]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 nonuniformly sized cells as well, their performance may deteriorate for reasons not previously considered.

[15]At the time when this paper goes to print, an improved version of ten-start FBB is available that performs much better on larger circuits. In the simpler implementation ten source/sink pairs are randomly generated and FBB is called ten times. For each source/sink pair (s', t') if the partitioning result (which separates s' and t') does not separate the given pair of fixed vertices (s, t), the result is illegal and cannot be used. The best result among the legal solutions is returned. The better implementation merges s' with s and t' with t before calling FBB. In this way, all ten trials produce legal solutions and the observed performance is considerably improved. Our correspondence with the authors of [19] indicates that the better FBB implementation may have a window of applicability, intermediate between end-case optimal partitioners and FM. However, we have been unable to confirm this due to platform incompatibilities as this paper goes to print.

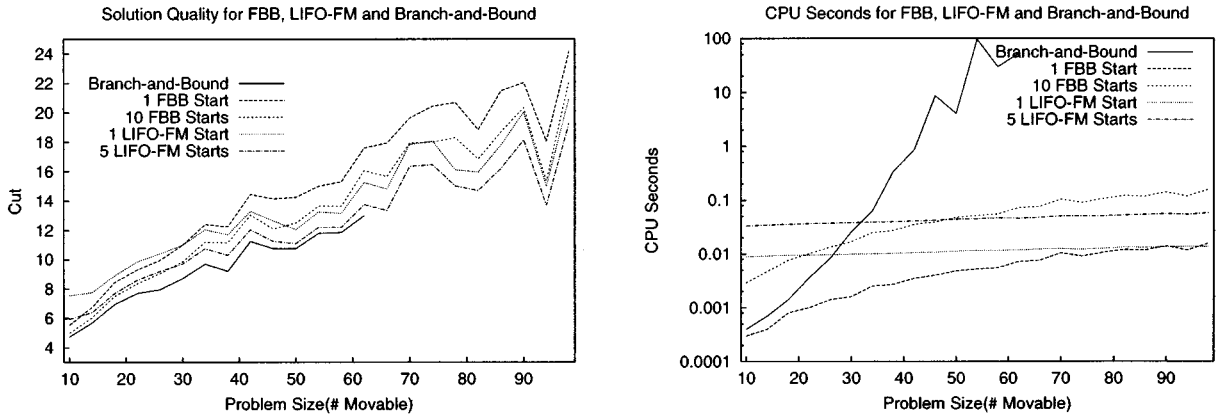


Fig. 9. Comparison of our Branch and Bound methods with one or ten starts of FBB and one or five starts of FM on problems saved from our placement tool. Data expressed as (average cut/average CPU time); time is expressed in CPU seconds on a 300-MHz Sund Ultra-10.

than 10%, while the run time of branch-and-bound is clearly acceptable for up to 30 nodes (<0.02 s). For instances with 30 nodes or more, FM provides better solution quality than FBB (as measured in one start) and needs fewer starts to achieve comparable quality. In fact, five starts of FM dominate ten starts of FBB for instances of 50 or more nodes. While FM and branch-and-bound do well in our comparison, they clearly have competition on small problem instances.[16] The run time plot in Fig. 9 also shows that FM scales very well (almost horizontal run time curves), FBB also scales well, but worse than FM. The branch-and-bound run time, as expected, scales exponentially (note the logarithmic scale of run time on the plot). The curves become more rugged for larger instance size because our pool of instances, generated from top-down placement, has fewer instances of large size making averaging less effective.

While our experiments were 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 netlist size while the same relative improvement in quality applies. Our top-down placer compares favorably to a recent release of a commercial (fixed-die, standard-cell) placer, and we believe that our reported improvements can carry over to other modern implementations.

More efficient branch-and-bound codes are undoubtedly possible and their study is the subject of ongoing research. To facilitate further collaboration in this direction, we have made the end-case partitioning and placement problem instances available through the Placement slot in the MARCO/GSRC Bookshelf for VLSI CAD algorithms [5]. Other important research questions include the use of multiway partitioners as well as alternative partitioning and placement objectives.

[16]All partitioning instances have a tolerance of at least 10%. If the tolerance is decreased, branch-and-bound has additional advantage, while FM performs much poorer (see our earlier results reported in [6]). FBB appears to perform only marginally worse.

REFERENCES

[1] C. J. Alpert. Partitioning Benchmarks for VLSI CAD Community. [Online]. Available: http://vlsicad.cs.ucla.edu/cheese/benchmarks.html

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

[3] C. J. Alpert, J.-H. Huang, and A. B. Kahng, "Multilevel circuit partitioning," in *Proc. ACM/IEEE Design Automation Conf.*, pp. 530–533.

[4] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *Integration*, vol. 19, pp. 1–81, 1995.

[5] A. E. Caldwell, A. B. Kahng, and I. L. Markov. MARCO/GSRC Bookshelf for VLSI CAD Algorithms. [Online]. Available: http://vlsicad.cs.ucla.edu/GSRC/bookshelf

[6] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Optimal partitioners and end-case placers for standard cell layout," in *Proc. IEEE Int. Symp. Physical Design*, Monterey, CA, 1998.

[7] J. Clausen and J. L. Träff, "Do inherently sequential branch-and-bound algorithms exist?," *Parallel Processing Lett.*, vol. 4, no. 1 and 2, pp. 3–13, 1994.

[8] 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 Trans. Electron Devices*, vol. 45, pp. 580–589, May 1998.

[9] A. E. Dunlop and B. W. Kernighan, "A procedure for placement of standard cell VLSI circuits," *IEEE Trans. Computer-Aided Design*, vol. 4, pp. 92–98, Jan. 1985.

[10] S. Dutt and W. Deng, "VLSI circuit partitioning by cluster-removal using iterative improvement techniques," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1996, pp. 194–200.

[11] S. Dutt and H. Theny, "Partitioning around roadblocks: Tackling constraints with intermediate relaxations," in *Proc. IEEE Int. Conf. Computer-Aided Design*, 1997, pp. 350–355.

[12] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions," in *Proc. ACM/IEEE Design Automation Conf.*, 1982, pp. 175–181.

[13] L. Hagen, J. H. Huang, and A. B. Kahng, "Quantified suboptimality of VLSI layout heuristics," in *Proc. ACM/IEEE Design Automation Conf.*, 1995, pp. 216–221.

[14] E. Ihler, D. Wagner, and F. Wagner, "Modeling hypergraphs by graphs with the same mincut properties," *Inform. Processing Lett.*, vol. 45, pp. 171–175, 1993.

[15] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Applications in VLSI design," in *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 526–529.

[16] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Tech. J.*, vol. 49, pp. 291–307, 1970.

[17] J. Kleinhans, G. Sigl, F. Johannes, and K. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 356–365, Mar. 1991.

[18] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. New York: Wiley-Teubner, 1990.

[19] H. Liu and D. F. Wong, "Network flow-based multiway partitioning with area and pin constraints," *IEEE Trans. Computer-Aided Design*, vol. 17, Jan. 1998.

[20] R. H. J. M. Otten, "Global wires harmful?," in *Proc. ACM/IEEE Int. Symp. Physical Design*, 1998, pp. 104–109.
[21] R. H. J. M. Otten and R. K. Brayton, "Planning for performance," in *Proc. ACM/IEEE Design Automation Conf.*, 1998, pp. 122–127.
[22] *The PARTY Partitioning-Library User Guide, Version 1.1*, Univ. Paderborn, Paderborn, Germany, Sept. 1996.
[23] H. D. Simon and S.-H. Teng, "How good is recursive bisection?," *SIAM J. Scientific Comput.*, vol. 18, no. 5, pp. 1436–1445, 1997.
[24] D. Stroobandt, "Improving Donath's Technique for Estimating the Average Interconnection Length in Computer Logic," *ELIS Tech. Rep.*, June 1996.
[25] L. Trotter, "PERM (Algorithm 115)," *Commun. ACM*, vol. 5, 1962.
[26] R. S. Tsay and E. Kuh, "A unified approach to partitioning and placement," *IEEE Trans. Circuits Syst.*, vol. 38, pp. 521–633, May 1991.
[27] J. Xu, P.-N. Guo, and C.-K. Cheng, "Cluster refinement for block placement," in *Proc. Design Automation Conf.*, June 9–13, 1997, pp. 762–765.

Andrew E. Caldwell received the B.S. degree in computer science from the University of California, Los Angeles (UCLA), and is currently pursuing the Ph.D. degree in computer science at the School of Engineering and Applied Science, UCLA.

In January 2000, he joined Simplex Solutions, Inc. Sunnyvale, CA, as a Senior Engineer. His research interests include VLSI layout and design, metaheuristic approaches for global optimization, and graph algorithms.

Andrew B. Kahng received the A.B. degree in applied mathematics (physics) from Harvard College, Cambridge, MA, and the M.S. and Ph.D. degrees in computer science from the University of California, San Diego.

After 11 years on the University of California, Los Angeles (UCLA) computer science faculty, most recently as Professor and Vice-Chair in the CSE and ECE departments as of January 2001. He was the founding General Chair of the ACM/IEEE International Symposium on Physical Design, and since 1997 has defined the physical design roadmap for the SIA International Technology Roadmap for Semiconductors. He currently leads the "C.A.D." theme in the MARCO Gigascale Silicon Research Center (http://vlsicad.cs.ucla.edu/GSRC/) and the U.S. Design Technology Working Group for the International Technology Roadmap for Semiconductors. He has published over 160 papers in the VLSI CAD literature, centering on physical layout and performance analysis. His research interests include VLSI physical layout design and performance analysis, combinatorial and graph algorithms, and stochastic global optimization.

Prof. Kahng received the National Science Foundation (NSF) Young Investigator award and a Design Automation Conference Best Paper award.

Igor L. Markov (M'97) graduated from Kiev University, Kiev, Ukraine, in 1993 and received the M.A. degree in pure mathematics from the University of California, Los Angeles (UCLA), in 1994. He received the Ph.D. degree in computer science from UCLA in 2000.

In 1995, he worked on solid modeling computer-aided design for Parametric Technology Corporation, Waltham, MA. Currently, he is an Assistant Professor in the Electrical Engineering and Computer Science Department at the University of Michigan, Ann Arbor.

Prof. Markov's research interests focus on CAD for VLSI physical layout including floorplanning, hypergraph partitioning, circuit placement, and more broadly, convex and combinatorial optimization.