An *O(n)* Algorithm for Transistor Stacking with Performance Constraints

Bulent Basaran and Rob A. Rutenbar

Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, PA 15213

Abstract

We describe a new constraint-driven stacking algorithm for diffusion area minimization of CMOS circuits. It employs an Eulerian trail finding algorithm that can satisfy analog-specific performance constraints. Our technique is superior to other published approaches both in terms of its time complexity and in the optimality of the stacks it produces. For a circuit with n transistors, the time complexity is O(n). All performance constraints are satisfied and, for a certain class of circuits, optimum stacking is guaranteed.

1 Introduction

In the layout of custom CMOS cells, *stacking* is defined as merging the diffusion regions of two or more transistors that have a common node, e.g., series-connected transistors have one node in common which can share a diffusion and save area. Since stacking has a dramatic impact on the total diffusion area and therefore on chip yield, there has been an extensive amount of research on optimizing leaf-cell layout through stacking. The original work of Uehara and van Cleemput [1] first posed the problem and offered a heuristic solution for digital circuits. For this important two-row P-over-N layout style, polynomial time algorithms were later discovered to arrange series-parallel dual CMOS ([2] is a good survey here). When more general aspects of the layout are to be optimized, e.g., wiring as well as stacking, a variety of combinatorial search algorithms have been used with success, e.g., [3].

In the analog domain, stacking is critical not only for area, but also for circuit performance due to parasitic diffusion capacitances. Unfortunately, the wider range of device sizes, and requirements for device matching and symmetry render the simpler row-based digital layout styles inadequate for analog. To address this, Cohn *et al*. introduced a free form 2-D stacking strategy integrated with device placement [4]. Charbon *et al.* later introduced a technique to satisfy performance constraints through constraint-driven stacking during placement [5]. Both tools can generate high-quality layouts, however, neither can *guarantee* a minimum diffusion area. More recently, [6] introduced a new stacking style and a novel technique to generate *optimum* stacks that satisfy performance constraints, using a path partitioning algorithm. However, because it attempts to enumerate all optimal stacks, runtime can be extremely sensitive to the size of the problem. Symmetry and matching constraints can greatly prune the search, but the basic algorithm has exponential time complexity [6].

In this paper, we present a new algorithm to perform stack generation in *linear* time. For a large class of circuits, our algorithm is *optimum* with respect to total diffusion area and a cost function modeling circuit performance. The cost function ensures that performance constraints are, if possible, met. Device matching is also guaranteed through symmetry and proximity constraints. The paper is organized as follows. Section 2 describes the basic stacking strategy. Section 3 explains how the circuit performance is modeled. In Section 4, the new stack generation algorithm is presented. Some results on industry-quality circuits are given in Section 4. Finally, Section 5 offers some concluding remarks.

2 Basic stacking strategy

A stacking methodology is needed to model the circuit schematic in a format appropriate for a graph algorithm to solve the layout problem effectively. Our strategy is similar to that introduced in [6] and earlier in [4] in more general terms:

1. Divide the circuit into *partitions* with respect to device type and bias node (body node in MOS transistors).
2. Perform device *folding*: split large transistors into smaller parallel transistors. These are called “fingers” by designers; we refer to these more generally as *modules* as they are the component pieces of our solution.
3. Perform further partitioning to reduce the variation on the module widths in a partition,
4. *Generate* stacks that implement each partition.

In analog CMOS circuits, as in digital standard-library leaf-cells, only transistors of the same type (e.g., NMOS), which share a common well, can be stacked (i.e., their common diffusion nodes can be merged in the layout to minimize diffusion area). In addition in analog circuits, it is fairly common to have transistors of the same type which require distinct body potentials, for example, to optimize noise performance. Such transistors have their own isolated wells and cannot be stacked with other transistors of the same type. Therefore in the first step, we put such transistors in different partitions. We also allow the designer to specify explicitly to have two or more transistors in the same stack.

In the second step, large transistors are folded into fingers to minimize the diffusion capacitances as well as to balance the aspect ratio of the resulting module. This can either be done automatically [6] or manually by the designer. It is important to note that, in this stacking strategy, transistor folding is done *a priori*. The stack generation algorithm is given fixed-width modules as input – it does not dynamically fold transistors. This is in contrast to tools such as KOAN [4], in which the overall optimization loop treats stacking, folding and placement simultaneously. Of course, such a separation of design tasks is sub-optimal. One of our main motivations in this paper is to devise a stacking strategy that is fast enough to be used in the inner loop of a placement tool like KOAN.

In the third step, the partitions are examined again to account for variations in module widths. If it is requested, modules with widths significantly larger, or smaller, than others in a partition can be put in a separate partition. This will result in a better utilization of space, but it will have suboptimal diffusion sharing. If such a partitioning is not acceptable for performance reasons, this step may be

33rd Design Automation Conference ®
Permission to make digital/hard copy of all or part of this work for personal or class-room use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 96 - 06/96 Las Vegas, NV, USA
©1996 ACM, Inc. 0-89791-833-9/96/0006.. $3.50

skipped.

In the fourth step, the stack generation algorithm (Section 4) operates on each circuit partition separately. We note that a pair of phases before and after the stacking algorithm may handle special patterns required by some analog circuits: module interleaving (i.e., common-centroid or inter-digitated device pairs); devices with ratio constraints to obtain precise current ratios (e.g., current mirrors) [6]; multi-fingered devices with proximity constraints [7].

3 Modeling performance constraints

During stack generation, it is required that certain performance specifications are considered and, if possible, met. The input to the stack generation algorithm (Section 4) is a cost function based on *criticality weights* on circuit nodes and *symmetry constraints* on the devices. In this section, we will briefly review how these parameters are obtained from performance specifications. Our approach follows [8] and [9].

The process of translating high-level circuit performance specifications into bounds on low-level layout parameters is called *constraint generation*. This process is traditionally done manually by circuit designers. Recently, techniques have been proposed to automate this process using sensitivity analysis [8].

Constraint generation starts with small signal sensitivity analysis of performance functions at the nominal operating point. Performance constraints are defined as maximum allowed variations of the performance functions around the nominal operating point. These constraints can be mapped to *parasitic capacitance constraints* on certain nodes and *matching constraints* on devices. The parasitic capacitance constraints, together with bounds on estimated parasitic capacitances, can further be translated into *criticality weights*, denoted *w*, on nodes. The tighter the constraints, the closer the minimum allowed performance to the estimated nominal value, the higher the weights. A cost function evaluating a stacking solution is introduced in [6] that minimizes the parasitic capacitance of critical nodes. We will use the same cost function to guide our stack generation algorithm. It is shown in Eq. (1) for the sake of completeness.

$$Cost(stacking) = \sum_{diff} w(diff) \cdot k(diff) \qquad (1)$$

Here, the summation is carried over all diffusion regions in the stacks. *w(diff)* denotes the criticality weight on the node that corresponds to *diff*. *k(diff)* is 1 if *diff* is a merged diffusion in the stacking. Otherwise it is given by C_{ext}/C_{int} where C_{ext} and C_{int} ($C_{ext} \geq C_{int}$) are the capacitances of an unmerged (external) and a merged (internal) diffusion, respectively. Note that when *w* is 1, the cost function minimizes only the total diffusion area. *w* is an effective way of prioritizing critical nodes during stacking.

Matching constraints are translated into symmetry constraints on devices and wiring and also to device proximity constraints. In order to match devices, our stack generation algorithm employs symmetry constraints on the devices of the circuit. The stacks obtained with a stack generation algorithm should be symmetric around a symmetry axis with respect to the twin transistors in them (Fig. 1) [14]. Further matching can be enforced earlier in the partitioning step of our stacking strategy as in [6] as well as later during placement and routing [4][15].

The next section describes in detail how the cost function in Eq. (1) is optimized and how the symmetry constraints are satisfied in the stack generation algorithm.

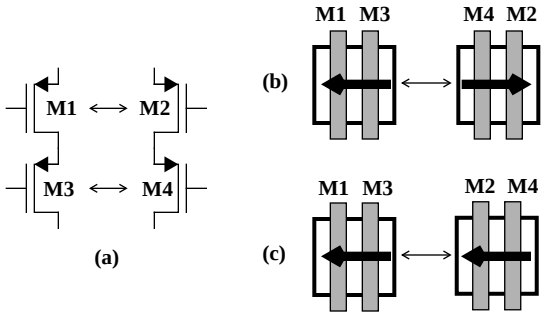

Fig. 1. Two symmetric transistor pairs (a) and their layout with symmetric stacks. Stacks in (b) and (c) are *mirror symmetric* and *perfect symmetric*, respectively.

4 Stack generation algorithm

As introduced in [1], finding an Eulerian trail in a diffusion graph is equivalent to minimizing the diffusion area of series-parallel static CMOS circuits. Later [10] presented a simple linear time Eulerian trail finding algorithm for dynamic CMOS circuits consisting of only one type of network (e.g., an nFET logic network). In our algorithm, we use a similar algorithm for finding an Eulerian trail. The main contributions of this algorithm are twofold:

1. *Performance*: We optimize a cost function that considers not only area but also circuit performance – this was previously achieved in exponential time [6],
2. *Generality*: Without any symmetry constraints, the algorithm is optimum. With symmetry constraints, it is still optimum for a large class of circuits.

Given a circuit partition, our algorithm first generates a *modified diffusion graph, G*, that represents the circuit partition. *G* incorporates the performance constraints in the form of criticality weights as defined in Section 3, as well as the symmetry constraints among transistors. Next a trail cover on *G* is found that satisfies the symmetry constraints in the circuit. In the final step each trail in the trail cover is converted to a transistor stack for layout. The outline of our algorithm is given in Fig. 2.

```
procedure stack(circuit_partition ckt)
1   generate the modified diffusion graph, G, from ckt
2   trail_cover = sym_trail_cover(G)
3   convert trail_cover into transistor stacks
4   return(transistor stacks)
```

Fig. 2. The stack generation algorithm.

Next we describe the modified diffusion graph and the symmetric trail cover finding step in detail and give an analysis of the algorithm.

A The modified diffusion graph, G

Let *ckt* be the circuit partition for which we wish to generate the transistor stacks. *ckt* can be represented with an undirected graph *G´* (possibly with parallel edges) called the *simple diffusion graph*. Each vertex in *G´* corresponds to a diffusion node in the circuit, and each edge in *G´* corresponds to a transistor (Fig. 3).

Let *v* be a vertex in *G´*; *v* is labeled with *w*(*v*) and *s(v)*. *w(v)* denotes the criticality weight on the node that corresponds to vertex *v*. *s(v)* denotes the symmetric twin of *v*. Let *e* be an edge in *G´*; *e* is labeled with *s*(*e*) = *e´*, where (*e*, *e´*) is a symmetric edge pair. *s*(*e*)=*e´* ⇒ *s*(*e´*)=*e*. Note that a diffusion graph with symmetry constraints

must be *fully symmetric*: all the edges must have symmetric twins. Otherwise, the circuit must be partitioned further so that each partition is fully symmetric (Fig. 4 (a)). A pair of symmetric edges are called *cross-symmetric*, if they cross the symmetry axis. A vertex is called *self-symmetric*, if $s(v) = v$. A self symmetric vertex is on the symmetry axis which cuts the graph into two halves (vertex *v7* in Fig. 4 (a)).

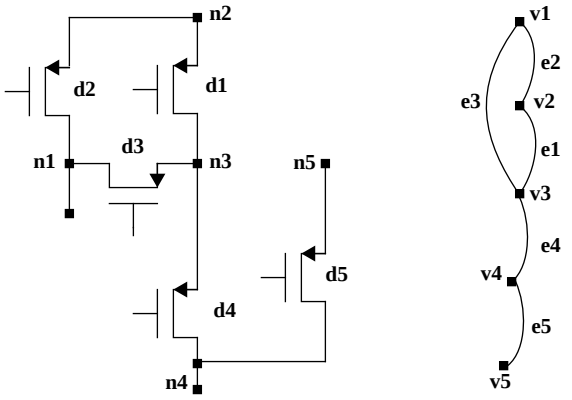

Fig. 3. A circuit partition and its simple diffusion graph. Each node in the circuit, *n*, is mapped to a vertex *v* in the graph; each device, *d*, is mapped to an edge *e*.

First we introduce some terminology from graph theory that will be used in the following sections. A *trail* on a graph is a set of edges $(v_0,e_0,v_1,e_1,v_2,..., v_{k-1}, e_{k-1},v_k)$, where $e_i=(v_i,v_{i+1})$ is an edge in the graph and $e_i \neq e_j$ for all $i \neq j$ [12]. We may use the shorthand $(v_0,v_1,v_2,...,v_k)$ or $(e_0,e_1,e_2,...,e_{k-1})$ to denote a trail. Note that an edge in a trail can not appear more than once but a vertex can appear at more than one position. Each such position is called a *terminal* of the vertex. v_k and v_0 are called the *end terminals* of the trail. The trail is a *closed* trail, if $v_k=v_0$.

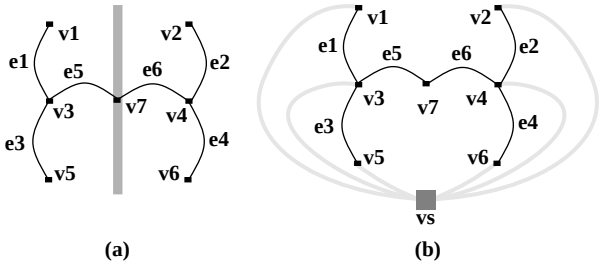

Fig. 4. (a) A simple diffusion graph with symmetry constraints. Note that pairs of edges with symmetry constraints are drawn symmetrically around the vertical symmetry axis. (b) The modified diffusion graph obtained from (a). Gray lines are the super-edges.

A set of trails, $T=\{t_i\}$, is called a *cover* for the graph if $\forall e \in G$, $\exists t_i$ s.t. $e \in t_i$ and $e \notin t_j$, $\forall j \neq i$. *T* is called a *minimum trail cover* if the number of the trails in *T*, or the cardinality of *T*, $|T|$, is the smallest among all possible set of trails. For example, for the graph of Fig. 4 (a), two trails *(v1,e1,v3,e5,v7)* and *(v5,e3,v3)* together with their symmetric twins *(v2,e2,v4,e6,v7)* and *(v6,e4,v4)* cover the whole graph. Let T_1 denote the set of these trails. Note that $|T_1| = 4$. Joining the first and the third trails at *v7*, their common end terminal, we can reduce the cardinality of T_1 to 3 which is the minimum for this graph.

A closed trail is an *Eulerian* trail, if it touches all the edges in the graph. A graph is called *Eulerian* if there exists a closed Eulerian trail on it. The degree of a vertex *v*, denoted *d(v)*, is the number of edges adjacent to it. It is well known in graph theory that a graph is Eulerian if and only if it is connected and all vertices in the graph have even degree [13]. Obviously in an Eulerian graph we can always find a trail cover T_1 with cardinality 1, since there is an Eulerian trail on it. It is also easy to see that in a graph that has n_{odd} vertices with odd degree, the minimum trail cover has a cardinality of $n_{odd}/2$ (It is known that n_{odd} is always even).

Note that in general the simple diffusion graph *G´* is not Eulerian. Let n_{odd} denote the number of vertices with odd degree in *G´*. If $n_{odd} > 0$, we add a vertex, called a *super-vertex*, v_s, to *G´* and we make it Eulerian by adding a new edge (v_s, v_i), called a *super-edge*, for each odd-degreed v_i. We set $w(v_s)$ to 0, since its criticality, by definition, is zero. The graph obtained from the simple diffusion graph, *G´*, by the addition of **(1)** the super-vertex and **(2)** the super-edges for odd-degreed vertices is called the *modified diffusion graph* and is denoted as *G* (Fig. 4 (b)).

B Finding a symmetric trail cover

If there are no symmetry constraints, we can find an Eulerian trail, t_e, in *G*, using a recursive Eulerian trail finding algorithm [13]. Let t_e be $(v_s,v_1,v_2,...,v_k,v_s)$. If we delete the super-edges in t_e, we obtain a set of trails, T_e, that has a cardinality of $n_{odd}/2$. Therefore, T_e is a minimum trail cover for *G´*, the simple diffusion graph.

However, when there are symmetry constraints, an arbitrary Eulerian trail, in general, does not yield a feasible solution. Here, we propose an algorithm which can be used to find a minimum trail cover in the presence of symmetry constraints. Our symmetric trail cover algorithm employs the same recursive algorithm for finding an Eulerian trail with modifications to handle perfect and mirror symmetry constraints. The outline of the algorithm is given in Fig. 5.

The algorithm `sym_trail_cover()` starts by selecting the vertex, v_0, with the lowest criticality weight. Next it finds a set of trails, *cover_left*, in Line 2 with the call to the recursive procedure `euler`() (Fig. 6). Here we note that the first trail `euler()` generates, *first_trail*, has v_0 at its end terminal; more on this in Section C. The trail cover, *cover_left*, including *first_trail*, covers only half of the edges in the modified diffusion graph, since at each iteration of Line 10 in `euler()` we not only delete the edge that is inserted in the trail but also its symmetric twin.

```
procedure sym_trail_cover(G)
1   pick v0 s.t. w(v0) ≤ w(vi) for all i≠0
2   first_trail= euler(v0)     // inserts open trails in cover_left
3   insert first_trail in cover_left
4   remove the super-edges at the end terminals
5   join_trails(cover_left)
6   if there are symmetry constraints
7       foreach trail tr in cover_left
8           construct the symmetric trail tr´
9           if tr and tr´ have a common end terminal
10              join tr and tr´ at the common end terminal
11              insert the result into cover_all
12          else
13              insert tr and tr´ into cover_all
14  decompose all the trails by deleting the super-edges
15  return(cover_all)
```

Fig. 5. Finding a symmetric trail cover.

In Line 5 of `sym_trail_cover()`, the procedure `join_trails()` concatenates the open trails in *cover_left* at their end terminals if possible (Fig. 7). This step is required due to the existence of cross-symmetric edges in the modified diffusion graph.

```
recursive procedure euler(vertex vin)
1   if d(vin) = 0      // no edges
2       return vin     // trivial trail
3   // starting from vin create a random trail tr:
4   vtemp = vin
5   do
6       if d(vtemp) = 0;
7           break;     // open trail
8       insert vtemp into tr
9       pick an edge on vtemp, e=(vtemp, vneigh)
10      delete e, s(e), if exists, from G
11      vtemp = vneigh;
12  while vtemp ≠ vin      // iterate until a closed trail is found
13  let tr = (vin,v1,v2,...vk)
14  find tr2 = euler(vin),euler(v1),euler(v2),...,euler(vk)
15  if vtemp = vin      // closed trail
16      return concatenation of tr2 and vin: (tr2, vin)
17  else      // open trail
18      insert tr2 into t_cover
19      return vin
```

Fig. 6. Finding an Eulerian trail with symmetry constraints.

```
procedure join_trails(cover_left)
1   if there is only one trail in the list
2       return      // no pairs to join
3   foreach trail tr=(v1,...,vk) in cover_left
4       let tr=(a,...,b)      // a and b are end terminals
5       insert tr in list(a) and list(b)
6   foreach end terminal x
7       join trails in list(x) pair-wise at x
8       update effected list
9   return(cover_left)
```

Fig. 7. Joining open trails.

Next, in Line 7-Line 13, symmetric twins of the trails in *cover_left* are constructed. This is possible, since as a trail in *cover_left*, *tr*, was being generated in `euler()`, the edges required to construct its symmetric twin, *tr´*, were preserved by deleting them from the graph. This process can also be viewed as simultaneously generating two trails that traverse the two halves of the graph in a synchronous and symmetrical way. Line 8 can construct either a mirror symmetric trail or a perfectly symmetric trail. In Line 9-Line 10 the trail *tr* and its symmetric twin *tr´* are joined, if they have a common end terminal and if the operation does not violate a perfect symmetry constraint. Fig. 8 shows an example.

As a consequence of deleting both of the edges in a symmetric pair, `euler()` may encounter a vertex with a zero degree while it is trying to find a closed trail in the `do-while` loop, Line 5-Line 12. When such a vertex is reached, `euler()` detects that the current trail has to be an open trail. For an open trail, `euler()` first recurses on the vertices of the open trail, as is the case with closed trails, but when the recursion terminates, it inserts the open trail in the trail cover *cover_left* and returns the initial vertex as a trivial trail to the previous recursion level (for more details and some examples see [18]). Note that in an Eulerian graph without symmetry constraints there is always a closed trail; no open trails are detected and `euler()` returns an Eulerian trail.

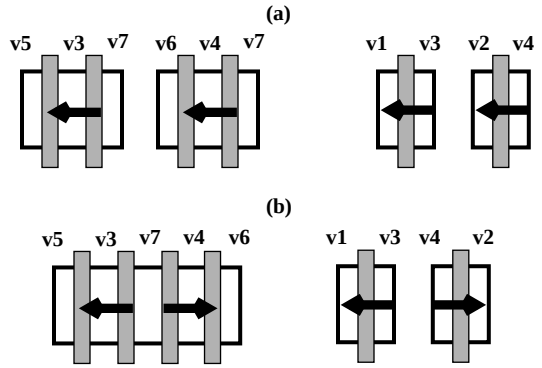

Fig. 8. (a) Perfect and (b) mirror symmetric trail covers; T_a={(v5,v3,v7), (v6,v4,v7), (v1,v3), (v2,v4)}, T_b={(v5,v3,v7,v4,v6), (v1,v3), (v4,v2)} and the corresponding stacks for the graph of Fig. 4 (a).

C Analysis of the algorithm

Time-complexity: The `do-while` loop in Line 5-Line 12 of `euler()` encounters each edge of the graph at most once, therefore it has complexity *O(n)*, where *n* denotes the number of edges. The two `foreach` loops in `join_trails()` operate on each trail only for a constant number of steps. Hence the complexity is *O(m)*, where *m* denotes the number of trails. But since *m = O(n)*, the complexity of `join_trails()` is *O(n)*. It follows that the overall complexity of the algorithm is *O(n)*.

Optimality: If there are no symmetry constraints, it is easy to see that the algorithm minimizes the cost function defined in Eq. (1): `euler()` returns an Eulerian trail which is later decomposed by deleting the super-edges (if any). Let us assume that the trail cover has k trails after the decomposition; $k = \max\{1, n_{odd}/2\}$. Also note that every vertex v in G must have at least $\lceil d(v)/2 \rceil$ terminals in a trail cover T. First assume $n_{odd} > 0$. If $d(v)$ is odd, then v has $d(v)/2 + 1$ terminals in the trail cover. Otherwise it has $d(v)/2$ terminals. In either case the number of terminals is equal to the lower-bound given by $\lceil d(v)/2 \rceil$. Now assume $n_{odd} = 0$. The previous argument still holds for all vertices except the one at the end terminals (Note that $k = 1$). But the vertex at the end terminal was chosen to be the one with the lowest criticality weight, therefore the cost function is minimized and the stacking is optimum. The cost function in Eq. (1) is also minimized for a class of circuits with symmetry constraints for which the corresponding modified diffusion graph satisfies two conditions: (1) no cross-symmetric edges (2) number of self-symmetric vertices with degrees $d(v) = 2(2k+1)$, $k \geq 0$ is less then 4. Given these conditions the optimum can be found in linear time by adding a post-processing step to the algorithm which recombines certain trails to reduce the cardinality further. The proof is rather long and will be presented in another paper. When the second condition is waived, the optimum can still be found via a similar post-processing step, but with a penalty in the time-complexity of the algorithm. Currently we are working on a sufficient condition for optimality in the general case.

It is also worth noting that we do not evaluate the cost function given in Eq. (1) in `sym_trail_cover()`. After stacking, the performance of the circuit can be evaluated using estimates on parasitic diffusion capacitances and device matching, looking at the generated stacks [6]. If there is an unsatisfied performance constraint, then the stack generation step indicates that the performance specifications were too tight and it is infeasible to meet them during the layout phase; hence either the design or the specifications must to be modified.

4 Results

The stack generation algorithm presented in this paper has been implemented in C++ on an IBM PowerPC 604 (133MHz) based workstation running AIX 4.1. We have tested the algorithm on various circuits from the literature.

Table 1 lists some of these circuits that we obtained from the literature [4][15][6] and shows some results. For all of the circuits the number of stacks is optimum and hence equal to the results obtained by [6]. Again note that the technique presented in [6] is enumerative and has exponential time complexity. We note that in theory our algorithm can guarantee optimality for only some classes of circuits. But still it could find the optimum results for all the circuits that were available to us, since most practical circuits indeed fall into the class for which our technique is proved to be optimum. Sensitive circuit nodes are maximally merged, and estimated performance degradation, as computed by Eq. (1), is equivalent to that in [6]. The run time is very low (less then 100ms per circuit). This compares favorably to [6] which employs an exponential-time algorithm; e.g., for *Comp3*, our optimum stack generation algorithm found a solution in less than 100msec while the technique in [6] reports 7.5sec, a difference of approximately two orders of magnitude[1]. For bigger circuits, higher savings can be expected.

Table 1. Stacking results.

Circuit	Ref.	# of devices	# of modules	# of ckt. partitions	# of stacks
Opamp1	[4]	10	22	2	4
Opamp2	[6]	29	32	5	9
Opamp3	[5]	11	30	3	3
Opamp4	[4]	27	40	3	11
Opamp5	[6]	25	36	9	10
AB	[6]	15	29	6	9
Comp2	[4,6]	15	25	4	5
Comp3	[6]	19	33	4	4
Mult	[15]	12	46	2	3
Buffer	[15]	10	53	2	4

Fig. 9 shows a multiplier circuit [15]. It is a typical analog circuit that was used as a benchmark in KOAN [4] as well as in other constraint-driven layout research [15]. The stacking solution generated with our algorithm is shown in Fig. 10. The number of stacks found is 3, which is the theoretical optimum. As a comparison, the number of stacks found in the KOAN layout is 7[2].

Fig. 11 shows another analog cell, a comparator which is highly sensitive to device mismatch and parasitic capacitance [4][15][6][16]. The stacking generated by our algorithm is shown in Fig. 12. Again, compared to KOAN, our algorithm found a better stacking, with 3 fewer stacks.

1. Also note that in [6], an enumerative algorithm is utilized which can find all optimum solutions whereas our technique finds only one.

2. We note that this is not a fair comparison, since KOAN integrates stack generation with placement.

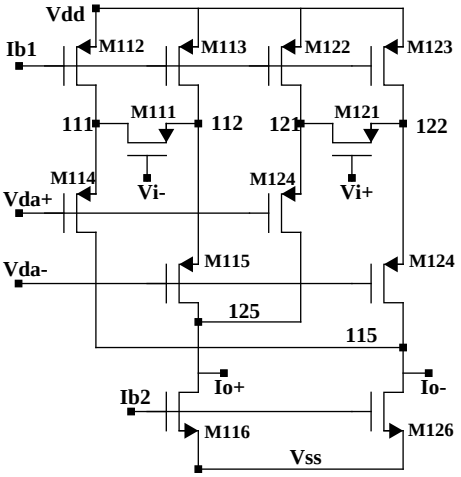

Fig. 9. The *Mult* circuit.

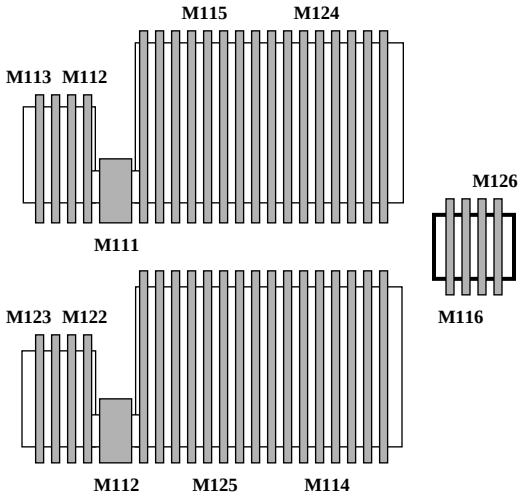

Fig. 10. The optimum stacking generated for *Mult*.

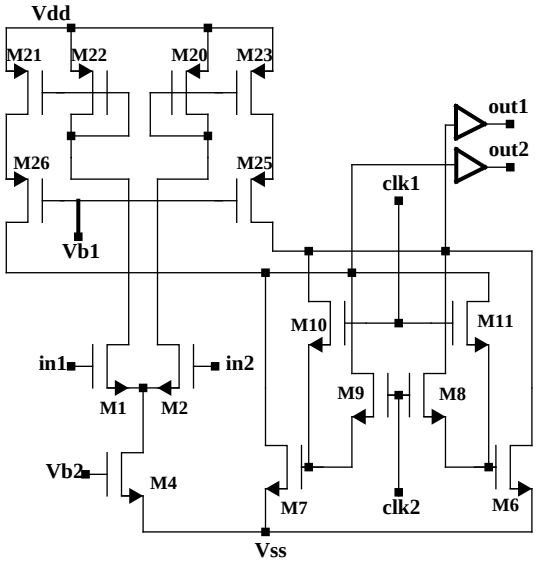

Fig. 11. The *Comp* circuit.

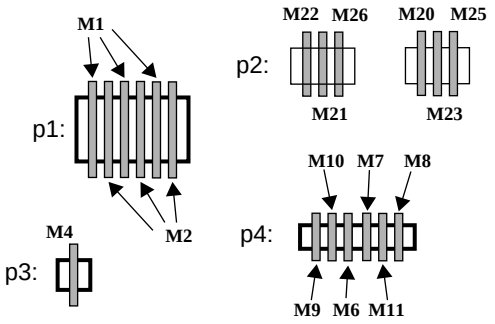

Fig. 12. The optimum stacking generated for *Comp*.

5 Conclusions

First-generation custom analog cell layout tools relied on simultaneous stacking, folding and placement of devices to achieve acceptable density and performance. The disadvantage of these approaches is the lack of any guarantees on the achievable circuit performance, and (due to their annealing-based formulations) the variability in layout solutions, run to run. Second-generation tools have focused on two-phase approaches, in which a partition of the devices into optimal stacks is performed first, and subsequent placement manipulates a palette of alternative stacks. The advantage is more predictable circuit performance, and these techniques can be fast for small circuits. But the runtime to generate all stack partitions can be extremely sensitive to circuit size due to the exponential algorithms at the core of these approaches. In this paper we introduced an effective stacking strategy that is fast enough to be exploited in the inner loop of a device placer, yet still respects analog node criticality information. In comparison with the 2-D free-form stacking style of [4], our approach is faster and can find better results. In comparison with the branch-and-bound technique of [6] which enumerates all optimum solutions, our approach can find a single solution of equivalent cost, for most practical circuits, but in linear-time with respect to the circuit size.

Our long term goal in this work is to integrate this stacking algorithm into a device placer in the style of [4], replacing random search for good merges with directed search among local clusters of devices. Instead of finding all stacking alternatives *a priori*, we only stack those local sets of devices that the placer tells us *ought* to be stacked. This should yield improved analog cell layout tools, and digital cell layout tools as well. Complex dynamic-logic CMOS cells are increasingly analog in character, and we believe that a combination of aggressive search (for device placement and folding) coupling with *simultaneous, dynamic* stacking proposed in [17] (to optimally arrange local clusters of devices) is an attractive strategy here.

Acknowledgments

We are grateful to Prof. Ron Bianchini and Pinar Keskinocak (CMU) for helpful discussions on Eulerian trails. We thank Prof. Rick Carley (CMU) and Dr. John Cohn (IBM) for giving us some of the circuits used in this paper. We thank Mehmet Aktuna for fruitful discussions. Pinar Keskinocak and Aykut Dengi also helped to improve the presentation by reading an earlier draft of the paper. We would also like to acknowledge MPI, Germany for their LEDA library which was of great assistance in prototyping with graph algorithms and basic data structures. This work is supported in part by the Intel Corporation and the Semiconductor Research Corporation.

References

[1] T. Uehara and W. M. vanCleemput, "Optimal Layout of CMOS Functional Arrays", *IEEE Transactions on Computers*, Vol. C-30, No. 5, May 1981, pp. 305-312.

[2] R.L. Maziasz, J.P. Hayes, *Layout Minimization of CMOS Cells*, Kluwer Academic Publishers, Boston/London, 1992.

[3] S. Wimer, R.Y. Pinter, J.A. Feldman, "Optimal Chaining of CMOS Transistors in a Functional Cell", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, September 1987, pp. 795-801.

[4] J. M. Cohn, D. J. Garrod, R. A. Rutenbar and L. R. Carley, "KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing", *IEEE Journal of Solid-State Circuits*, Vol. 26, No. 3, March 1991, pp. 330-342.

[5] E. Charbon, E. Malavasi, U. Choudhury, A. Casotto, A. Sangiovanni-Vincentelli, "A Constraint-Driven Placement Methodology For Analog Integrated Circuits", *IEEE Custom Integrated Circuits Conference*, May 1992, pp. 28.2/1-4.

[6] E. Malavasi, D. Pandini, "Optimum CMOS Stack Generation with Analog Constraints", *IEEE Transactions on Computer-Aided Design*, Vol. 14, No. 1, Jan. 1995, pp. 107-122.

[7] M.J.M. Pelgrom et al., "Matching Properties of MOS Transistors", *IEEE Journal of Solid-State Circuits*, Vol. sc-24, October 1989, pp. 1433-1440.

[8] U. Choudhury and A. Sangiovanni-Vincentelli, "Automatic Generation of Parasitic Constraints for Performance-Constrained Physical Design of Analog Circuits", *IEEE Transactions on Computer-Aided Design*, Vol. 12, No. 2, February 1993, pp. 208-224.

[9] E. Charbon, E. Malavasi, A. Sangiovanni-Vincentelli, "Generalized Constraint Generation for Analog Circuit Design", *Proceedings of the IEEE/ACM ICCAD*, Nov. 1993, pp. 408-414.

[10] S. Chakravarty, X. He, S.S. Ravi, "On Optimizing nMOS and Dynamic CMOS Functional Cells", *IEEE International Symposium on Circuits and Systems*, Vol. 3:, May 1990, pp. 1701-1704.

[11] S. Chakravarty, X. He, S.S. Ravi, "Minimum Area Layout of Series-Parallel Transistor Networks is NP-Hard", *IEEE Transactions on CAD*, Vol. 10, No. 7, July 1991.

[12] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, Elsevier Science Publishing, New York, 1976.

[13] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1982.

[14] J. M. Cohn, "Automatic Device Placement for Analog Cells in KOAN", PhD dissertation, Carnegie Mellon University, February 1992.

[15] B. Basaran, R. A. Rutenbar and L. R. Carley, "Latchup-Aware Placement and Parasitic-Bounded Routing of Custom Analog Cells", *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, November 1993, pp. 415-421.

[16] E. Charbon, E. Malavasi, D. Pandini, A. Sangiovanni-Vincentelli, "Simultaneous Placement and Module Optimization of Analog IC's", *Proceedings of the IEEE/ACM Design Automation Conference*, June 1994, pp. 31-35.

[17] B. Basaran and R. A. Rutenbar, "Efficient Area Minimization for Dynamic CMOS Circuits", *IEEE Custom Integrated Circuits Conference*, May 1996.

[18] B. Basaran and R. A. Rutenbar, "An O(n) Algorithm for Transistor Stacking with Performance Constraints", *Research Report No. CMUCAD-95-56*, Carnegie Mellon University, 1995.