Fast Isomorphism Testing for a Graph-based Analog Circuit Synthesis Framework

Markus Meissner, Oliver Mitea, Linda Luy, Lars Hedrich
Electronic Design Methodology, Department of Computer Science,
University of Frankfurt/Main, Germany
Email: {meissner, mitea, luy, hedrich}@em.cs.uni-frankfurt.de

***Abstract*—This contribution presents a major improvement for our analog synthesis framework with an explorative characteristic. The presented approach in principle allows the synthesis of a wide range of circuits, without the limitation to specific circuit classes. Defined by a specification of up to 15 different performances, a fully sized, transistor level circuit is synthesized for a provided process technology. The presented work reduces the needed computational effort and thus drastically reduces the synthesis time, while adding new abstraction into the framework to provide an even wider range of synthesized circuits - demonstrated in experimental results.**

I. Introduction

Although analog modules often take only a small fraction of the chip area on a modern integrated circuit, usually this modules demand the most engineering effort and thus a lot of development time of specialized and skilled designers. The ongoing development in integrated process technology - not only towards shrinking the structures - but also by developing new concepts as 3D-structures or even carbon nanotubes, further add new challenges to keep track with the always present time-to-market pressure. Compared to digital design, the analog synthesis flow has a low degree of automation. Whilst recent approaches to parameter synthesis, or sizing have shown the industrial usefulness of automated analog synthesis tools, the topology synthesis still lacks industrial awareness due to non existent, usable tools. This contribution aims to provide a methodology, which could fill this gap, by providing a fully automated flow for synthesizing the topology and sizing of analog circuits with given industrial specifications.

II. Our Contribution

The presented work is divided into two major parts. The first one focuses on the generation of unique circuits during synthesis to optimally utilize the available resources, as the methodology exhibits the inherent property to generate duplicates of circuits. This is the trade-off introduced by using basic blocks instead of single transistor synthesis, which already explodes in complexity by a transistor count of four. By significantly reducing the design-space through the usage of basic blocks instead of single transistors, the generation of duplicates has to be accepted and thus targeted. Whereas the second improvement to the framework is designated to the quality of the synthesized circuits, by adding a new device type and by further generalizing the methodology.

- An isomorphism algorithm handling a huge number of circuits was developed and further optimized to scale well on huge circuit counts. Further unique requirements introduced by an explorative analog synthesis flow have also been targeted.
- Capacitors, this leads to compensated circuits generated by the framework.
- Further generalization of the constructive synthesis rules to support an arbitrary number of signal paths.

III. Related Work

There was a lot of early work about general graph isomorphism [1] and it was made clear that the graph isomorphism is in NP, but not necessarily NP nor P-complete. Although practical implementations of the algorithm behave quite well. One of the first approaches was the Gemini algorithm proposed by Ebeling et al., for which later was shown that there are graphs for which the algorithm fails [2]. Further improvements developed by Ebeling and Weisfeiler [3] are now the base for many different general graph isomorphism algorithms as *nauty* [4]. An extensive overview about common isomorphism techniques and methods can be taken from [5].

This contribution presents various improvements to the topology synthesis methodology initially shown at [6] and [7]. At this point the methodology was undergoing a redesign to fully rely on a graph-based circuit representation and generation [8].

IV. Analog Circuit Synthesis Framework

The presented work aims to be a flexible analog synthesis framework based on basic building blocks as shown in Figure 1. Those building blocks are well known analog primitives taken from standard literature [9] as also used in [10]. Those blocks are grouped together as abstract basic blocks. Whereas each abstract basic block group contains only basic blocks, which share in principle the same input and output characteristics. Those characteristics are described through ports, each of them provides the following properties:

- Input or output port
- Voltage or current port with high or low impedance
- Negative or positive bias current direction (if applicable)

This work was partly developed within the project SyEnA (project label 01 M 30 86) which is funded within the Research Program ICT 2020 by the German Federal Ministry of Education and Research (BMBF).

978-3-9810801-8-6/DATE12/©2012 EDAA

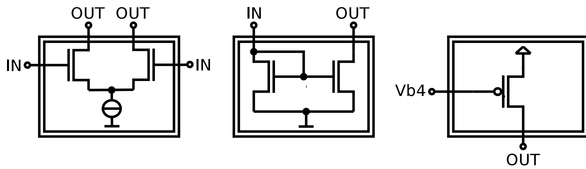


Fig. 1. Some example analog basic blocks

A. Graph-based Synthesis

The synthesis target is defined by a specification containing:

- Up to 15 performances (Gain, SlewRate, Area,...)
- Maximum allowed block count
- Input and output characteristics

To synthesize a topology, abstract basic blocks are connected together according to various constructive rules:

- Only interconnect voltages and correctly directed currents. Connect voltages only to high and currents only to low impedance ports. (**Elementary-Electric-Rule**)
- A current source or sink can be inserted to handle the affected node as a voltage node. (**Current-Source-Rule**)
- Two correctly directed currents can be connected to create a new node, which can also be handled as a voltage node. (**Current-Combine-Rule**)

The graph-based synthesis now generates topologies by repeatedly applying constructive rules to the available abstract basic blocks. Each generated topology that does not contain more than the maximum block count and fulfills the specified input and output characteristics is kept for further processing. To finalize the generation process, the generated topologies have to be expanded. This expansion is necessary, as the abstract basic blocks are just groups of basic blocks - like black boxes without a specific implementation. Circuit expansion takes place either in a symmetric or asymmetric way. Same abstract basic blocks are never expanded with two differing basic blocks, if symmetric expansion is applied. Opposing to asymmetric expansion, which expands each abstract basic block with all its variants, thus leading to lot more circuits. Throughout this paper symmetric expansion is used, further details about the expansion method can be taken from [7].

After expansion, the resulting circuits are checked for isomorphism (a destructive rule), which is described in detail in the following section. This finishes the circuit generation step as shown in Figure 2.

All generated circuits now get a bias circuit attached and are enqueued for evaluation. The *Circuit Database*, as shown in Figure 2, asynchronously assigns the tasks for symbolic analysis to an arbitrary number of specialized application servers, which communicate with minimal traffic over TCP/IP to provide the most flexible scaling possibilities for the synthesis framework. Once a symbolic analysis for a circuit results in a good performance, the circuit is enqueued for sizing on another type of application servers with WiCkeD [11]. More details about the symbolic analysis and sizing process can be taken from [7]. Finally the synthesis process returns fully sized transistor level circuits, which meet the specified performances.

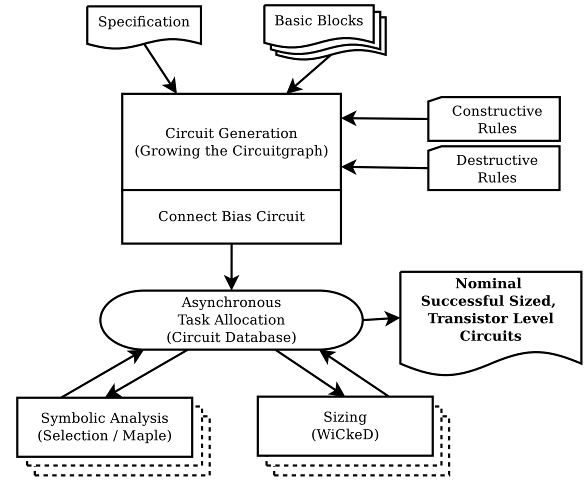


Fig. 2. Synthesis Flow

B. Compensation

The framework was extended to support a wider range of operational amplifiers and thus improving the generated circuit quality and flexibility. Capacitors can now be used and are evaluated throughout the whole generation and sizing process. This allows the placement of capacitors inside basic blocks for circuit generation and furthermore the capacitance is used as a design variable during the sizing, including a process specific area calculation. Initially two basic blocks with an included capacitor are provided to the synthesis process as shown in Figure 3.

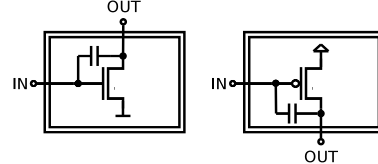


Fig. 3. Two basic blocks with miller capacitors

C. Arbitrary number of signal paths

The constructive Elementary-Electric-Rule and the Current-Source-Rule have been extended to support an arbitrary number of signal paths during circuit generation. This further generalizes the synthesis framework and allows to utilize the interconnection of basic blocks that have multiple input and output ports. In order to take advantage of this improvement, new basic blocks, as seen in Figure 4 and their pMOS counterparts, were added.

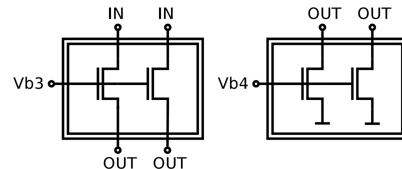


Fig. 4. *DualCascode* and *DualCurrentSource* blocks

V. Graph Isomorphism For Generated Circuits

Checking for isomorphism between two graphs is in general an NP-hard [5] problem, which would lead to an exponential complexity. Furthermore, the circuits have to be compared

pairwise, this means that the isomorphism algorithm has to be applied up to $O(|C|^2)$ times, with C being the total number of generated circuits. Additionally the term isomorphism has to be extended in the context of circuit synthesis.

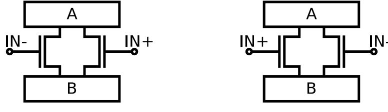


Fig. 5. Isomorphic circuits with swapped pins

Figure 5 shows two circuits with identical subcircuits denoted as A respectively B. The only difference between both circuits are the swapped pins. From the graph point of view they are indeed isomorphic, but as the generated circuits are analyzed, simulated and optimized fully without human interaction, it is necessary to classify the circuits in Figure 5 as not isomorphic. This guarantees that no two circuits are not classified as isomorphic, if their only difference is an unmatched pin.

A. *Defining a circuit as a graph*

A circuit c consists of i nets and j devices i.e. mos-transistors, capacitors or any other component:

$$N := \{\text{net}_1, \text{net}_2, \ldots, \text{net}_i\} \quad (1)$$

$$D := \{\text{device}_1, \text{device}_2, \ldots, \text{device}_j\} \quad (2)$$

these are always strictly connected to the opposite type through edges:

$$E := \{e_1, e_2, \ldots, e_n\} \quad (3)$$

Thus the circuit is represented as a bipartite graph using (1), (2) and (3):

$$c = (N, D, E) \quad (4)$$

Finally the generated circuits for a given specification S using a set of basic blocks B and a rule set R will be denoted as:

$$C = \text{gen}\,(S, B, R)$$

B. *An Isomorphism Algorithm for Huge Numbers of Circuits*

The used isomorphism algorithm is based on the Gemini algorithm described in [3], including the improvements suggested in [12], although it contains various modifications to make it applicable to the requirements given by the representation of synthesized analog circuits.

The primary idea is to create a graph partition P_i for a circuit c_i,

$$P_i := \{s_1, s_2, \cdots, s_k\} \quad (5)$$

which consists of k disjunctive subsets s_i containing all graph nodes:

$$\forall s, s' \in P_i \;\rightarrow\; s \cap s' = \emptyset \quad (6)$$

The number of nodes d inside a subset s_i is denoted as $|s_i|$, while a subset of size one is a *singleton*. To check two circuits c_1 and c_2 for isomorphism, the partitions P_1 and P_2 are simultaneously refined to contain only singletons. Partitioning is done implicitly by assigning a label to each node - identical labeled nodes are inside the same subset. A label of a specific node x is referenced as $L(x)$ throughout this paper.

$$s_i := \{\forall n, n' \in N \cup D \;:\; L(n) = L(n')\} \quad (7)$$

Initial partitioning is done by labeling the device D and net nodes N according to their node invariant. This node invariant for device nodes is the type of the device and for net nodes it is the degree of the node. Furthermore, if a device node has a connection to a pin of the circuit, the node invariant for this device node is altered depending on the pin it is connected to and the edge which connects it to the pin.

In practice this leads to a partition, where most pMOS devices are inside one subset, most nMOS devices are in another subset and net nodes inside subsets according to their degree.

The initial partitioning is one of the main improvements compared to the Gemini approach. Instead of assigning labels according to the device type only, the label is modified if the device is connected to nets, which represent a pin. So devices which are equally connected to one or more pins are handled in a separate subset. In other words: If both circuits contain one such device, those devices are matched from the beginning and obviously, if only one of the circuits contains such a device, those circuits cannot be isomorphic.

The algorithm consists of a preprocessing phase and the main phase, during which the graphs are pairwise checked for isomorphism. Algorithm 1 describes the full algorithm for a given $spec$, a set of basic blocks B and rules R. Additionally the *fulliso()* function is sketched out in Algorithm 2.

Algorithm 1 Unique circuits generator

```
C ← gen(spec, B, R)
DB ← ∅
∀c ∈ C : preprocess(c)
for all c1 ∈ C do
  accept = true
  for all c2 ∈ DB do
    if ∀p ∈ PR : p(c1) = p(c2) ∧ fulliso(c1, c2) then
      accept = false
      break {identified c1 as isomorphic}
    end if
  end for
  if accept then
    DB ← c1 {add new unique circuit c1}
  end if
end for
```

The $preprocess()$ function takes a circuit as input, extracts m properties

$$PR := \{prop_1, prop_2, \cdots, prop_m\} \quad (8)$$

and caches them for fast access during the main phase of the algorithm. Once the preprocessing is finished, the main phase starts. All generated circuits are checked for isomorphism

against all circuits in the database. If no isomorphism can be proved, the circuit is added to the database. Since first all properties of the two circuits are compared, it is not necessary to execute the full isomorphism for each pair of circuits. Only if all properties, extracted during preprocessing, are equal, the full isomorphism algorithm has to be executed. This method differs from the Gemini approach and speeds up the process and is analyzed in the results section.

Algorithm 2 $fulliso(c_1, c_2)$

```
P_1 ← initialPartitioning(c_1)
P_2 ← initialPartitioning(c_2)
Q ← {(s_1, s_2) : ∀s_1 ∈ P_1 ∧ ∀s_2 ∈ P_2 : min(|s_i|)}
{smallest subsets, assuming only one node |s_1| = |s_2| = 1}
while Q ≠ ∅ do
   node_1, node_2 ← Q.pop()
   next_1, next_2 ← getNeighbors(node_1, node_2)
   for all n_1 ∈ next_1 ∧ n_2 ∈ next_2 do
      if n_1 ∈ N_1 ∧ n_2 ∈ N_2 then
         L(n_i) = L(n_i) + Σ_{e∈E(n_i)} L(D(e))
      else
         L(n_i) = L(n_i) + Σ_{e∈E(n_i)} L(N(e)) · L(e)
      end if
      Q.push((n_1, n_2))
   end for
   if not singletonsMatch(P_1, P_2) then
      return false
   end if
end while
return true
```

With $E(x)$ representing the set of edges connected to the node x, and $D(y)$ or $N(y)$ representing the set of devices respectively nets connected to the edge y. $L(x)$ as previously denoted, returns the current label of the passed node x. Q has to be a queue to go over the graph with a breadth-first-search.

The $fulliso()$ function primarily implements the Gemini algorithm [3] to determine whether two circuits are isomorphic. First the smallest available subset of the partition is chosen - it always exists for both circuits as the partitions were checked to be equal. At this point it is assumed that both subsets only contain one node, so the algorithm starts at this node in both circuits.

For this node all neighbors are acquired and enqueued for further analysis, because of the bipartite nature of the graph, these are only net nodes for a device node and vice versa. For each neighbor of both circuits n_1 and n_2 now a new label $L(n_i)$ is calculated.

The node is now deleted from its original subset and added to the new subset determined by the newly computed label. On continuation the partitions are checked for singletons, these will be marked so that they will not be relabeled again. Once the number of singleton subsets differ for both circuits, the isomorphism check returns a negative result. Now the process is repeated for the previously enqueued nodes until all nodes are relabeled and all subsets are singletons. For symmetric circuits the case occurs that, after all enqueued nodes are processed, still subsets exist, which are not singletons. The nodes from these subsets are again enqueued and the process is repeated again.

It can easily be shown that, after refining both partitions to only contain singletons, c_1 is isomorphic to c_2.

$$\text{if } \forall s \in P_1 \ \wedge \ \forall s' \in P_2 \ : \ s = s' \ \wedge \ |s| = 1 \quad \rightarrow c_1 \text{ isomorphic to } c_2 \tag{3}$$

It was previously assumed, that the initially chosen subsets only contain one node each. Obviously the smallest subset may contain more than one node. For this case the isomorphism is not disproved, if the partition equality check fails after relabeling. This process must be repeated for all pairs of nodes from the initial subsets to disprove or prove isomorphism.

C. Accelerating the algorithm

With a rising number of circuits n the algorithm starts to struggle with the complexity, due to its $O(n^2)$ nature. The circuit count rises exponential with the number of maximum blocks, what even adds another class of complexity. So a very important optimization for speed up was applied to the algorithm.

The target is to reduce the $O(n^2)$ complexity - this could be achieved by reducing the number of circuits the algorithm has to compare a new circuit with. A well known algorithm for the general isomorphism problem is the *nauty* [4] algorithm. It is based on calculating a canonical representation of a given graph. This would allow a runtime of $O(n \cdot \log n)$ in our approach, if the representation would be used as a hash and all circuits could be stored inside a tree with the key being the hash. Unfortunately this would introduce a major disadvantage: the calculation of this canonical representation takes exponential time. This would lead to an explosion of time inside the preprocessing step.

Inspired by the idea of a hash-able representation of a circuit, a hash h is computed for every circuit c during preprocessing based on the previously calculated properties from (8):

$$h = \prod_{p \in PR} p(c) \mod \left(2^{32} - 5\right)$$

The number $2^{32} - 5$ is the largest prime number representable by a 32-bit unsigned integer and thus spans up a prime residue class, which behaves better on multiplicative operations to avoid collisions. This seems to be the best trade-off between simplicity of calculation and avoiding collisions.

As h is obviously not sufficient to verify isomorphism, at least equality is necessary for two circuits to prove isomorphism, because all properties have to be the same for two isomorphic circuits. So this hash is used to store circuits with the same hash inside a ordered tree structure. This allows to search for circuits with a given hash in $O(\log n)$, this reduces the overall complexity of the proposed pin-aware isomorphism algorithm for big amounts of circuits to a worst case runtime of $O(n \cdot \log n)$. The tremendous speed up this approach introduces to the algorithm is evaluated in the results section.

VI. Results

This section evaluates the contributions to the methodology. Starting with an analysis of the performance of the isomorphism algorithm and an evaluation of the efficiency of the hashing approach to reduce the complexity and thus the runtime of the algorithm. Followed by an evaluation of the synthesized circuits, which are classified into types of operational amplifiers. Finally the improvements to extend the range of synthesized circuits is analyzed.

Throughout this whole evaluation an Intel Core2 Quad 2.4Ghz server with 8GB RAM was used to generate the results. The analog synthesis framework synthesizes single-ended operational amplifier circuits based on a 180nm process technology and if not specifically mentioned the maximum block count is four. The software is implemented in C++ on a Linux platform, with Maple [13] to apply symbolic analysis and the commercial tool WiCkeD [11] to size the circuits using the Spectre simulator from the Cadence Suite [14] as simulation backend - all together in a fully automated flow.

A. Performance of the isomorphism algorithm

The unique requirement for the isomorphism algorithm inside this analog synthesis framework is to perform well, even for thousands of circuits. The isomorphism algorithm designed to fulfill these requirements, strongly relies on pre-calculated circuit properties. The analysis how often a specific property triggers and thus avoids the two circuits are being passed to the full isomorphism check, shows that more than 99% of the circuit isomorphisms can be decided through property comparisons.

For the results generated in Table I and II, a synthesis run with compensation, dual current sources, and dual cascode blocks was used. The maximum number of allowed blocks was set to three, four and five. Table I shows the absolute numbers of comparisons, generated/unique circuits and *fulliso()* checks done during the comparisons.

Max Blocks	Generated Circuits	Unique Circuits	Circuit Comparisons	fulliso() runs	Iso. runtime
3	192	88	16.928	136	0.68s
4	1.744	658	1.392.346	2.074	28.42s
5	16.112	4.643	114.339.702	23.753	2245.94s

TABLE I
Absolute Comparison Counts

One can clearly see that the chosen properties perform quite well on their task to avoid unnecessary full isomorphism runs. But unfortunately also the quadratic complexity is obvious, leading to a very high count of comparisons and thus runtime, what can be seen in Figure 6. This was, as mentioned in the previous section, the reason to generate a hash out of the properties and store circuits according to their hash inside a tree data structure. The absolute numbers after applying this improvement and repeating the synthesis process, can be taken from Table II.

The reduction of done comparisons is enormous and directly correlates with the runtime (Figure 6). Interestingly the number of comparisons does not match the number of of applied

Max Blocks	Generated Circuits	Unique Circuits	Circuit Comparisons	fulliso() runs	Iso. runtime
3	192	88	136	136	0.42s
4	1.744	658	2.074	1.930	3.57s
5	16.112	4.643	33.347	23.753	38.22s

TABLE II
Absolute comparison counts for hash method

isomorphism runs. The presented hashing algorithm focuses on a fast calculation of the hash, so as a trade-off, the number of collisions is higher, compared to a real hashing algorithm. These collisions can be calculated by subtracting the number of full isomorphism runs from the total comparisons. In practice they do not change the runtime significantly, because they are quickly discarded as they are still checked for all properties. This reduction of comparisons and runtime could be confirmed in all runs presented in this results section and shows itself as very stable and reliable.

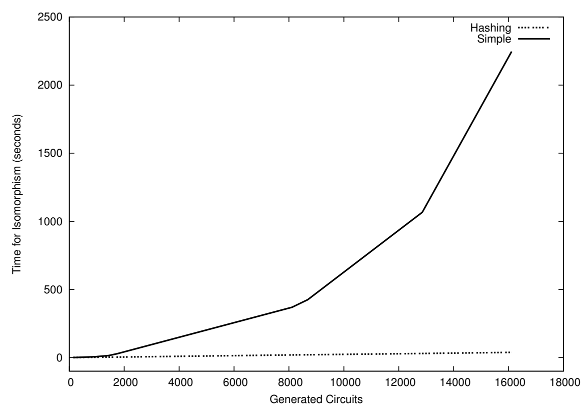


Fig. 6. Absolute isomorphism runtime vs. circuit count

Finally a runtime comparison of the simple and the hash-driven isomorphism algorithm is presented in Figure 6. The drastic increase of runtime renders the simple algorithm useless starting at some thousands of circuits. The additional time taken by the hashing algorithm is barely visible. The quadratic behavior of the simple algorithm is clearly visible, but also that the hash based algorithm nearly has a linear runtime. Nearly, because there is overhead introduced by the preprocessing and other administrative steps.

B. Synthesis Results

The isomorphism algorithm now allows a detailed analysis of the circuits being synthesized by the framework. Table III shows the generated and unique circuit count for different configurations of the synthesis. Compared to our prior contributions [8], one can see, that the generated number of circuits was 976 for four blocks, which is now represented by the (*SIMPLE*) run without compensation and arbitrary signal paths. Other presented configurations are: included compensation (*COMP*), compensation, dual current sources and dual cascodes all together (*COMP2CS+CASC*) and finally a run with only a maximum of three blocks (*3BLOCK*). To benchmark and compare the different runs, a quite easy specification, as seen in Table IV, was chosen to generate many nominal successful circuits.

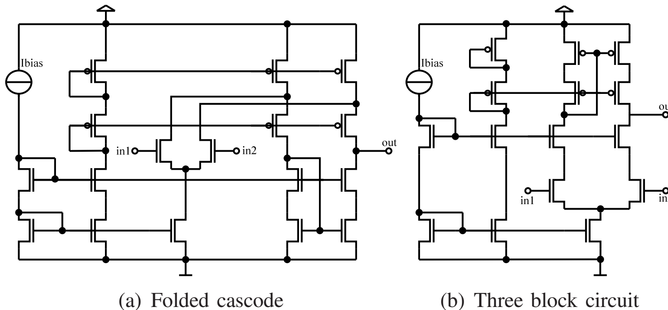
Configuration	Generated Circuits	Unique Circuits	Full Runtime	Nominal Successful
SIMPLE	976	361	2h 46min	17
COMP	1024	385	3h 08min	22
COMP2CS+CASC	1744	658	4h 55min	31
3BLOCK	192	88	23min	5

TABLE III
GENERATED/UNIQUE CIRCUITS FOR DIFFERENT CONFIGURATIONS

The efficiency and thus runtime win introduced by the isomorphism algorithm can be directly seen by comparing the time for the *SIMPLE* run with our prior work [8], which state a runtime of below 24 hours - this time was undercut by the factor of eight and can be further decreased by adding more application servers to evaluate circuits.

C. Analyzing The Synthesized Circuit Classes

First the *SIMPLE* run is analyzed, as this one is directly comparable to our prior work. Table III shows how nearly $\frac{2}{3}$ (63%) of the circuits were identified as isomorphic. The 17 nominal successful, unique circuits can be classified into seven OTAs (4x pMOS input stage and 3x nMOS input stage), two (uncompensated, nMOS input stage) Miller-OpAmps and eight quite unusual circuits. Except for the circuit shown in Figure 7(b) and Table IV the *3BLOCK* run only returns four additional unusual circuits.



(a) Folded cascode (b) Three block circuit

Fig. 7. Two synthesized circuits

The occurrence of only two Miller-OpAmps already gives a hint about the lack of a correctly placed and sized compensation capacitance. This drawback was eliminated by adding compensated blocks to the basic block library as seen in Figure 3. This addition to the library generates not only, the previously mentioned, two nMOS input stage Miller-OpAmp with compensation. Furthermore, now three Miller-Opamps with pMOS input stages are also successfully sized and made available.

As presented in Figure 4, to demonstrate the usefulness of the arbitrary signal path extension, so-called dual-blocks were added into the library to support an even wider range of circuits. Most notably this leads to the synthesis of folded-cascode operational amplifier topologies. In detail there are two folded-cascode circuits with an nMOS input stage as seen in Figure 7(a) including its performances in Table IV and also one folded-cascode circuit with a pMOS input stage. Moreover six, again very strange circuit topologies, are generated, which can not be clearly identified as a classic circuit from circuit design literature.

Name	Specifications	Fig. 7(a)	Fig. 7(b)
Area	$< 5000\mu m^2$	$2420\mu m^2$	$2920\mu m^2$
Gain	$> 50dB$	$51.41dB$	$66.74dB$
OutputVoltRange	$> 2V$	$2.13V$	$2.21V$
Offset	$-10mV \cdots 10mV$	$7.98mV$	$4.37mV$
Ft	$> 4MHz$	$4.09MHz$	$4.07MHz$
Power	$< 2mA$	$612\mu A$	$566\mu A$
PhaseMargin	> 45 deg	74.4 deg	73.5 deg
SlewRateFall	$> 1MV/s$	$1.56MV/s$	$1.63MV/s$
SlewRateRise	$> 1MV/s$	$1.58MV/s$	$1.54MV/s$
PSRR	$> 50dB$	$51.4dB$	$57.55dB$
CMRR	$> 50dB$	$58.8dB$	$51.42dB$
OvershootRise	$< 30\%$	7.53%	6.52%
OvershootFall	$< 30\%$	5.18%	6.15%
SettlingRise	$< 1\mu s$	$228ns$	$363ns$
SettlingFall	$< 1\mu s$	$245ns$	$375ns$

TABLE IV
SPECS AND THE CIRCUITS' PERFORMANCES FOR A 180NM TECHNOLOGY

VII. CONCLUSION

By further improving the presented graph-based, explorative analog synthesis methodology it is now possible to synthesize an operational amplifier circuit for a given specification in under three hours. This leads to a 8 times speed up compared to our prior work. More complex circuits with up to 30 transistors could now be synthesized overnight. The designer's task is now, to choose from the synthesized circuits to find a topology that fits to his requirements. This allows the exploration of many different topologies at once, instead of building and sizing a topology by hand, which usually results in one topology, without taking other (maybe better) topologies into account.

REFERENCES

[1] M. Goldberg, *The graph isomorphism problem, Handbook of graph theory, Discrete Mathematics and its Applications, chapter 2.2, pages 6878.* CRC Press, 2003.

[2] R. Mathon, "Sample graphs for isomorphism testing," *Congressus Numerantium, 21:499517*, 1978.

[3] C. Ebeling, *Gemini II: A Second Generation Layout Validation Tool*. In Proceedings of the IEEE International Conference on Computer Aided Design (ICCAD-88), 1988.

[4] B. D. McKay, "The nauty page," *Computer Science Department, Australian National University, 2004 http://cs.anu.edu.au/bdm/nauty/*, 2004.

[5] J. L. L. Presa, "Efficient algorithms for graph isomorphism testing," Ph.D. dissertation, Licenciado en Informatica Madrid, 2009.

[6] X. Wang and L. Hedrich, "An approach to topology synthesis of analog circuits using hierarchical blocks and symbolic analysis," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. IEEE Press, 2006, p. 705.

[7] O. Mitea, M. Meissner, and L. Hedrich, "Automated Constraint-driven Topology Synthesis for Analog Circuits," in *Proc. of the Conference on Design, Automation and Test in Europe*, 2011.

[8] M. Meissner, O. Mitea, and L. Hedrich, "Graph-based framework for explorative topology synthesis of analog circuits," in *FAC11*, 2011.

[9] B. Razavi, *Design of analog CMOS integrated circuits*. McGraw-Hill, Inc. New York, NY, USA, 2000.

[10] H. Graeb, S. Zizala, J. Eckmueller, and K. Antreich, "The sizing rules method for analog integrated circuit design," *ICCAD '01*, 2001.

[11] MunEDA GmbH, "www.muneda.com."

[12] M. Ohlrich and C. Ebeling, "SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm," *DAC '93: Design Automation Conference*, pp. 31–37, 1993.

[13] Maplesoft, "www.maplesoft.com."

[14] Cadence Design Framework, "www.cadence.com."