Parametric Hierarchy Recovery in Layout Extracted Netlists

John Lee, *Student Member, IEEE,* Puneet Gupta, *Member, IEEE,* and Fedor Pikus, *Member, IEEE,*

***Abstract*—Modern IC design flows depend on hierarchy to manage the complexity of large-scale designs. However, due to the increased impact of long-range layout context on device behavior and modeling thereof, extraction tools tend to flatten these designs. As a result, in post layout extraction the hierarchy is lost which leads to an increase in both the size of the design database, and the amount of runtime that is needed to process these designs. In this paper, the idea of parametric hierarchy recovery is proposed that takes netlists extracted from the design layout, and recovers their hierarchical structure while preserving parametric accuracy. This decreases the size of the netlist and enables the use of hierarchical comparison methods and analysis. Our experiments show that in physical verification this method leads to a 70% reduction in runtime on average without any parametric error. Furthermore, this method can be used to provide a tractable timing and power analysis that utilizes detailed transistor information in the presence of systematic layout-dependent variation.**

I. INTRODUCTION

Modern large digital designs are highly hierarchical: they are composed from standard cells which are organized into blocks, larger macros, and even larger chiplets or tiles. The hierarchical nature of these designs is used at every stage of the design and verification flow – it is used for the partitioning of the design to enable large design teams to work efficiently and used by most EDA tools to increase their performance by improving data and memory management (c.f. [1], [2], [3]). This results in orders of magnitude runtime improvements and in addition, it makes verification and debugging easier.

However, as the design proceeds through verification and characterization, the original hierarchy may degrade. The main reason is that at every stage the design is annotated with extra information which is necessary for the next stage but cannot be accurately represented within the original hierarchy. For designs targeting advanced manufacturing nodes, one of the main culprits of hierarchy degradation is the computing of complex layout-dependent device parameters, usually done during the circuit extraction. These device parameters describe the impact of different effects, such as stress, annealing, etch and lithographic variability, on device performance. Many of these are very long-range effects whose effective radius can exceed the size of the small hierarchy blocks. The resulting device parameters cannot be represented within the original hierarchy, since every placement of a cell would have slightly different parameter values.

P. Gupta and J. Lee are with the Department of Electrical Engineering, University of California at Los Angeles, CA, 90095 (e-mail: puneet@ee.ucla.edu; lee@ee.ucla.edu).

F. Pikus is with the Mentor Graphics Corporation, 8005 SW Boeckman Rd., Wilsonville, OR 97070 (e-mail: fedor_pikus@mentor.com).

The usual approach to this problem is to flatten the hierarchy whenever the parameters are different by "pushing" the transistor instance up the hierarchy. This creates an unnecessarily large number of replicas of the instance. However, such hierarchy degradation has significant negative impact on the downstream tools, since they can no longer take advantage of the design repetition and must process the design in a largely flat manner. The work of the designer is made much more complex as well: where at an earlier stages, one error might be reported in a standard cell, millions of errors might be seen across the entire chip, all with slightly different values, with no obvious way to detect that all these errors have a common origin and can be addressed by fixing the problem in the original cell.

The hierarchy degradation induced by device parameter variations represents real systematic variability phenomena and cannot be avoided altogether. However, these variations depend mostly on the layout context of each device. While the effective dimensions of this context may be much larger than the size of a cell or even a small block, the repetitive nature of the design implies that the number of significantly different contexts is usually fairly limited. We can reasonably expect that while devices in different placements of the same cell have different parameters, these differences can be represented without flattening the design completely if we introduce multiple variants of the original cell, each with different parameters. Of course, the success of this approach hinges on the assumption that the number of such variants is much smaller than the total number of cell placements.

The hierarchy recovery we just described can largely counteract the adverse effects of hierarchy degradation. For example, consider how it can be used to provide timing estimates for a design. It may be impossible to run a timing analysis tool on a design with a different model for each cell when the number of cells is in the millions. In these cases, it is attractive to map each cell instance to a representative cell variant (e.g. AND2X1 version1, etc.) and use the representatives to compute the timing estimate. A similar approach can be used for power estimates.

Another motivating example comes from the case of Layout Versus Schematic (LVS) verification and parasitic extraction, where the schematic netlist is compared to the polygon layout. In this process, the layout is first extracted as a transistor-level netlist. However, the original hierarchy of the layout cannot be retained in the extracted netlist due to device parameter variations. However, if the extraction process would utilize variants of the original cell, instead of flattening the cells down to device level, the resulting netlist could be very hierarchical.

The LVS circuit comparison tools can then utilize this netlist and set up a many-to-one correspondence between all variants of a layout cell and the corresponding schematic cell. Not only would this result in great runtime and memory improvements for the circuit comparison, but the LVS results would be much easier to debug since the errors are reported in their original cells (possibly with the list of variants to which the error applies). Parasitic extraction, Electrical Rule Checking (ERC), circuit simulation, and logical netlist analysis tools can realize similarly significant performance improvements compared to running on a flattened layout.

In this paper we explore the idea of parametric hierarchy recovery in the context of layout extracted netlists. Here, parametric refers to the parameters in the netlists, such as gate length, width and stress information. The idea is to recover the hierarchy in the flattened netlists as a means to manage the complexity, enable faster runtimes, and extract information about the designs. We give algorithms for recovering hierarchy, and show applications of this method in physical verification and static timing and power analysis.

II. Recovering Hierarchy

In the hierarchical design process, the designs are created by using building blocks of parent-types $\mathcal{T}_{\text{parent}} = \{A, B, C, D, ...\}$. Each of these types are instantiated as instances $\mathcal{I} = \{a_1, ..., b_1, ..., c_1, ...\}$. For example, instance a might be, by design, of type NAND2X1, and instance b might be an 8-bit adder block.

As the design process progresses towards fabrication, the hierarchy becomes flattened as additional parameters are added to each instance, increasing the runtime for any tools that use the resulting designs. However, the hierarchy can be recovered with respect to the different parameters. *Type-variants* ($\mathcal{T}_{\text{variant}}$) can be used to improve the accuracy of the parent-type. For example, type-variants NAND2X1a and NAND2X1b may be created to cover different variations. NAND2X1a may be an instance with a 2% increase in l_{eff} and a 3% increase in v_{th}, while NAND2X1b may be an instance with corresponding *decreases* in l_{eff} and v_{th}. Each instance is then mapped to an expanded set of variants to increase the accuracy of the type / instance correspondence.

The mapping between instances to types can be formalized using a *type mapping* function ($\text{type}(\cdot)$) that matches each instance with a unique type. $\text{type}_{\text{parent}}(\cdot)$ matches each instance to its parent-type, and $\text{type}_{\text{variant}}(\cdot)$ matches each instance to its assigned type-variant. There are many possible functions, and the proper choice of mapping functions depends on the available types. Instances that are mapped to the same type are said to be *clustered*.

The error related to a type mapping depends on the application – in timing-related applications, the parameters related to I_{sat} would be weighted heavily, while in power related applications, the I_{off} would be weighted heavily. In layout verification applications, the error would be related to the worst-case values over all of the instances, and also related to the amount of tolerance that is needed to create a match. We will discuss the measures of error in Section III-A.

III. Methods for Type Mapping and Type Creation

Type mapping and type creation are very similar to the *clustering* problem. In the clustering problem, instances are grouped together into clusters, to minimize an error function between each member of the cluster and the cluster centroid[1]. Similarly, in the context of type mapping, the objective is to minimize some measure of the error (see Section III-A) which is a function of the difference between the instance's parameters and the type-variant's parameters.

A. Error measures

In this work, we consider two different classes of error objectives. The first class is related to applications in power. In this case, we use the square error measure of the error:

$$\text{Error}_2(\mathcal{I}) = \sum_{i \in \mathcal{I}} ||p_i - p_{\text{type}_{\text{variant}}(i)}||^2 \tag{1}$$

where p_i is a vector of parameters associated with instance i and $p_{\text{type}(i)}$ is a vector of parameters associated with the type associated with instance i. Note that this error measure is continuous and differentiable. As an intuitive measure of the size of the error per transistor, the following are also used in this paper:

$$\mu_p = \text{mean}_i\{p_i - p_{\text{type}_{\text{variant}}(i)}\} \tag{2}$$

$$\sigma_p = \sqrt{\text{mean}_i\{(p_i - p_{\text{type}_{\text{variant}}(i)})^2\}}. \tag{3}$$

The second class of error objectives is related to timing and verification, and is neither continuous nor differentiable. This error is a *tolerance* objective:

$$\text{Error}_{\text{tol}}(\mathcal{I}) = \begin{cases} \infty & \text{if } |p_i - p_{\text{type}_{\text{variant}}(i)}| > p_{\text{tol}} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

In the above, p_{tol} is a given tolerance for the parameters. In other words, this error function is ∞ (e.g. it is unacceptably large) if *any* of the parameters exceed the tolerance, and 0 otherwise. In this case, the objective is to find a mapping that does not violate the tolerance.

B. Algorithms for Type Mapping

Once the number of type-variants or the tolerance is chosen, a suitable clustering algorithm is needed to group together different instances into clusters, and create type-variants to represent each cluster. In the context of layout extracted netlists, the number of instances of each type-parent is in the 100's of thousands, and may run into the millions. In this case, it is important to consider *scalable* algorithms. The following methods are used to perform type mapping in this paper.

1) k-Means algorithm: The k-Means algorithm [5] is a popular algorithm for performing clustering. It scales well and can be used to cluster millions of instances under the Error_2 objective. In this algorithm, the number of clusters is given, and the assignment to the clusters is refined iteratively:

* At iteration n, instances $\mathcal{I}$ and type-variants $\mathcal{T}^{(n)}_{\text{variant}}$

[1]See [4] for an overview of clustering methods

1) Refine the mapping from instances to variants ($\text{type}_{\text{variant}}(\cdot)$) by:
   a) Assign each instance i to the type variant $\tau \in \mathcal{T}^{(n)}_{\text{variant}}$ that minimizes the Error_2
2) Create updated type-variants $\mathcal{T}^{(n+1)}_{\text{variant}}$ by choosing the parameters of $\mathcal{T}^{(n)}_{\text{variant}}$ to minimize Error_2

Step 1(a) is performed by exhaustive search – each instance is compared against all available type-variants, and the one with the minimum error is chosen. This assumes that the number of clusters is generally much smaller than the number of instances, which is true in the context of hierarchy recovery. Step 2 is performed by taking the expected value of the instances in each type-variant.

2) Clustering with tolerance: The k-Means algorithm cannot be used effectively when a tolerance-based clustering is needed. This is because the derivative of the objective $\text{Error}_{\text{tol}}$ is constant ($= 0$) wherever it is defined. Furthermore, clustering with tolerance will give the number of clusters as an output, while the number of clusters is an input to k-means.

In this paper, tolerance based clustering is performed using a simple heuristic that assigns each instance to the first matching type-variant:

* **Start:** with all instances $\mathcal{I}$ unclustered, and empty set of type-variants ($\mathcal{T}_{\text{variant}} = \{\}$)
* **Do:** For each instance i
   a) Assign each instance i to the first $\tau \in \mathcal{T}_{\text{variant}}$ that satisfies the tolerance condition (e.g. $\text{Error}_{\text{tol}} = 0$)
   b) If no type-variant exists within the tolerance, create a new type-variant, and add the type-variant to $\mathcal{T}_{\text{variant}}$

3) Hierarchical type mapping with tolerance: In many designs, the instances may be composed of smaller, lower-level instances, with multiple levels of hierarchy. For example, consider the case of a controller block that is composed of fifos, which are in turn composed of flip-flops. These cases are ubiquitous in modern VLSI designs as the hierarchy is essential for managing the complexity of the design.

The hierarchy complicates the creation of types, as the higher-level types must be composed of the available lower-level blocks types in $\mathcal{T}$. Furthermore, higher-level types that are mapped to the same variant must use the same corresponding variants for the lower-level types. Thus there is an interaction across the levels of the hierarchy that must be considered.

Figure 1 shows an example of a hierarchical design. The design undergoes hierarchical type-mapping in Figure 2. In this example, the higher-level clustering *induces*, or forces, instances at the lower-level to be clustered together. These clusters must be maintained when the lower-level instances are clustered together.

A variation of the algorithm in Section III-B2 is used to perform hierarchical type mapping:

1) Sort the parent-types $t_k \in \mathcal{T}_{\text{parent}}$ such that each t_k is not composed of types t_j, $j < k$.
2) Looping over k, cluster the instances of each parent-type t_k by:

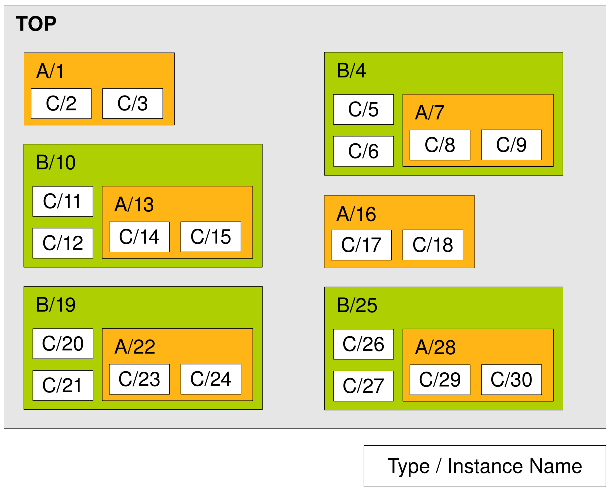


Fig. 1. Example of a hierarchical design. The design currently has types A, B, and C with 30 instances total. Instances in $\{1, 7, 13, 16, 22, 28\}$ are of type A; instances in $\{4, 10, 19, 25\}$ are of type B; the remainder are of type C.

   a) Creating initial clusters that are induced by higher level parent-types
   b) Checking each pair of clusters, and merging them if they are jointly within the tolerance
   c) Merging instances into the existing clusters if possible, otherwise create new clusters

Note that each instance has a unique parent-type associated with it. Also, when there is an ambiguity in the merging due to multiple matching clusters (steps (2a) or (2b)), the cluster or instance is added to the first cluster that meets the tolerance.

Figure 2 shows an example of this process. The method starts at the top of the hierarchy (with the B parent-type), clustering the instances into two type-variants (B1 and B2)[2]. The next level of hierarchy is then processed (the A parent-type). In this case, there are initial clusters that are induced (formed) from the mapping into type-variants B1 and B2. These clusters are used as an initial set of clusters (step (2a)), and then each pair of clusters are checked for possible merging with other clusters by checking the pairwise tolerance (step (2b)). Lastly, the free instances that are not yet assigned are merged into existing clusters if they fall within the tolerance. If they do not meet the tolerance of any cluster, they are assigned to a new cluster.

IV. Applications

A. Experiment 1: Validation

Post-layout validation requires a Layout Versus Schematic comparison, where the layout is validated against the original netlist. This step checks if all of the devices in the original netlist are present in the layout, and if the connectivity in the netlist is accurately represented in the layout.

This process can be thought of in two steps. First a netlist is extracted from the layout that has transistor information and the connectivity information between transistors. Next,

[2]Note that steps (2a) and (2b) are skipped at the top level of hierarchy, as there are no higher level parent types.

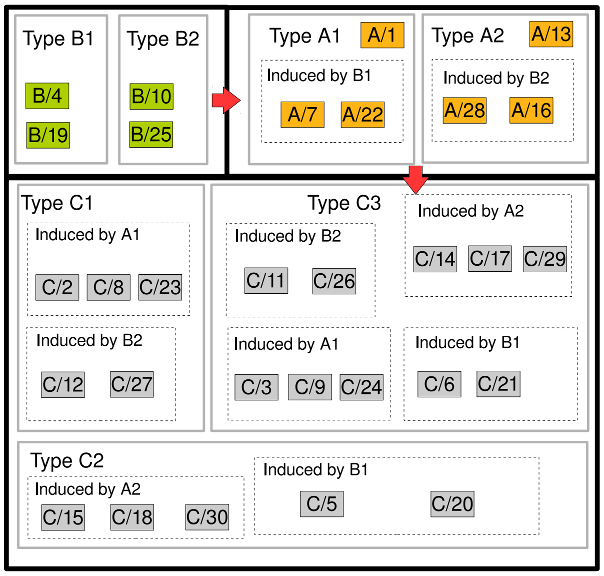


Fig. 2. Top-down clustering example for the design in Figure 1. The clustering at the higher levels *induces* a clustering at the lower levels. In this example, clustering instances B/4 and B/19 induces the clustering of {A/7, A/22 }, {C/6, C/21}, and {C/5, C/20}. These induced clusters can be merged with other induced clusters and instances, as in the creation of type A1

the extracted netlist is compared against the original netlist to verify that the two netlists are the same.

In the netlist extraction process, however, extracting the complete set of parameters flattens the hierarchy. This is because each instance of a master cell (such as an "AND" gate) gains extra properties due to its neighboring structures, such as stress, lithography effects, and coupling capacitances. This increases the runtime of the verification[3].

Currently, traditional LVS tools may attempt to partially recover the hierarchy, but are limited because they do not create variants of the original cells to preserve the exact values of extracted properties. These tools may employ a *push* process that starts at the top level of the hierarchy, and when it encounters devices which are of the same parent-type and have identical parameters in all placements, the tool performs a *pushdown*, whereby the devices are pushed down the hierarchy. In this manner, the hierarchy increases, and can be partially recovered. However, as a drawback, the devices in the hierarchical cells, such as NAND2X1, may be split across different levels of hierarchy. This splitting causes huge slowdowns in verification, as the transistors need to be flattened for comparison. Moreover, this flattening makes the debugging process much tougher. Thus, preserving and/or recovering the hierarchy will enable validation procedures to run much faster.

For example, suppose that the ADD64 block has the following sets of transistors that correspond to the M0 of NAND2X2:

```
ADD64: NAND2X2: M0 PARAM1=1
ADD64: NAND2X2: M0 PARAM1=1
ADD64: NAND2X2: M0 PARAM1=1
```

In this case, the M0 transistor of NAND2X2 will be pushed down out of the ADD64 block, and into the NAND2X2 cell. However, suppose that the M1 transistors of NAND2X2 has the following parameters:

```
ADD64: NAND2X2: M1 PARAM1=1
ADD64: NAND2X2: M1 PARAM1=1
ADD64: NAND2X2: M1 PARAM1=1.001
```

In this case, because the third transistor does not match, the push down is prevented and the transistor remains in the ADD64 block. However, this means that *every* M1 transistor in *every* NAND2X2 will not be able to be pushed down, creating as many extra instances of the M1 transistor as there are instances of NAND2X2, which may be in the hundreds of thousands.

However, the example above could be handled with type creation. Instead of pushing the transistors into the same type, two different types could be created, such as:

```
NAND2X2_A M1: PARAM1=1
NAND2X2_A M1: PARAM1=1
NAND2X2_B M1: PARAM1=1.001
```

This has two benefits. First, the type creation preserves the entire cell in the hierarchy, and allows the verification tools to run faster. Second, this may result in a smaller netlist than the pushed netlist.

As an experiment, the hierarchy recovery method is applied to five *real* industrial circuits designated "test0", "test1", "test2", "test3", "test4". The test1, test2 and test3 circuits are Miltie-core versions of the same design – test1 is a 1x1 core die, test2 is a 3x3 die and test3 is a 4x4 die. Test 0 has 8 parameters that vary between instances of the same type, tests 1-3 have 13 parameters that vary, and test4 has 39 parameters that vary. These parameters are related to transistor context information, and transistor stress information. Note that in these examples, *there is no variation in the gate lengths and gate widths*.

The recovery is performed with a:

- 0% tolerance (denoted as $R_{0\%}$),
- 10% tolerance (denoted as $R_{10\%}$),

where the percentage tolerance is with respect to the maximum deviation of the parameter. Thus, if parameter "EMX" has a *maximum deviation* between instances of $5 \cdot 10^{-6}$, then a $R_{10\%}$ would set the tolerance to be $0.5 \cdot 10^{-6}$. The sizes of these circuits are summarized in Table I which lists the number of transistors in the total design, the fully hierarchical netlist, the pushed netlist, recovered netlist with zero tolerance ($R_{0\%}$), and the recovered netlist with a tolerance of 10% ($R_{10\%}$).

Table I shows that the number of transistors in the recovered netlist with zero tolerance is smaller than the total transistors, and smaller than the number of transistors in the pushed netlist, showing that the hierarchy recovery is more effective at modeling the variations. In these examples, the difference in size between the recovered netlist and the pushed netlist grows as the size of the circuit grows. Also, the recovery is very effective in the test2 and test3 designs, where it can exploit the hierarchy that is inherent in the design. Figures 3 and 4 summarize the transistor count and the subcircuit count, as a function of the tolerance.

We measure the effect of hierarchy recovery on the netlist comparison process (LVS) using Calibre [6]. In this experi-

[3]Note that while ignoring layout context parameters may be ok for LVS, the extracted netlist is also used downstream by parasitic extraction tools.

TABLE I
EXPERIMENT 1- BENCHMARKS

	# of transistors (sub circuits)				
	flat	hierarchical	pushed	$R_{0\%}$	$R_{10\%}$
test0	17,505,258	15,051 (219)	2,330,409 (219)	92,236 (941)	91,604 (866)
test1	834,062	1,072 (368)	834,062 (368)	833,456 (61,108)	162,628 (13,170)
test2	3,336,248	1,072 (369)	3,256,313 (369)	833,468 (61,112)	162,628 (13,171)
test3	7,506,558	1,072 (369)	7,293,398 (369)	833,468 (61,112)	162,628 (13,171)
test4	2,354,107	4,485 (1219)	1,027,717 (1219)	83,405 (33,623)	32,614 (11,880)

ment, the pushed and recovered netlists are compared against the hierarchical netlist.The comparison runtime depends on the size of the netlist, and also on the list of hierarchical-cell pairs that is given to the verification engine. This list contains pairs of sub circuits, one from the layout and one from the source netlist, and instructs the tool to compare the elements of the pair hierarchically. In the LVS experiments, type-variants with 2 or more instances, and 1000 or more elements are added to this list using a bottom-up method (starting with the deepest level of the hierarchy). When counting elements in a type-variant, the size of the elements within the sub circuits is also counted, unless that subcircuit is on the hcell list.

The runtimes for this process and the runtime of the hierarchy recovery process are shown in Table II. The comparison time[4] is shorter than the pushed comparison time for all of the five $R_{0\%}$ examples. When the hierarchy-recovery time is accounted for, two of the benchmarks are slower. Note, however, that the slower comparisons (test0 and test3) benefit the most, and are faster with the hierarchy recovery. A plot of the runtime vs. tolerance and memory vs. runtime is shown in Figures 5 and 6. The plots show that the hierarchy recovery with any tolerance significantly reduces long LVS runtimes and never appreciably degrades the total runtime.

Another step of validation is the Electrical Rule Checking (ERC), which checks if the electrical requirements are correct by analyzing the netlist. For example, ERC can be used to find all PMOS transistors that have gates tied to VDD, or perform more complicated checks such as finding all inverter structures that have their gates tied to VDD. As the checks become more and more complex, the runtime difference between the pushed netlist and the clustered netlist will grow.

Table II and Figures 7 and 8 shows an example of an ERC run in Calibre [6] for a set of common checks[5]. Hierarchical-cell pairs are created for type-variants with 2 or more instances and 2 or more elements. All of the benchmarks run faster at the $R_{0\%}$ point, with an average runtime reduction of 27% and a minimum reduction of 11%. At the $R_{10\%}$ point, the average reduction is 63.4%, with a minimum reduction of 56%. The results show a signicant improvement in runtime that increases as the tolerance increases. The memory usage is not always better, and is worse for the test1 and test2 benchmark from $R_{0\%}$ to $R_{10\%}$.

[4]The hierarchy recovery time measures the amount of time that is needed to create type-variants and remap them, and excludes the time that is needed to read the data and prepare the data in memory. However, these times are also excluded from the netlist comparison times.

[5]The benchmark test0 was unable to run as the netlist had non-standard transistors. The runtime given is the total CPU time, minus the time needed to read the netlist.

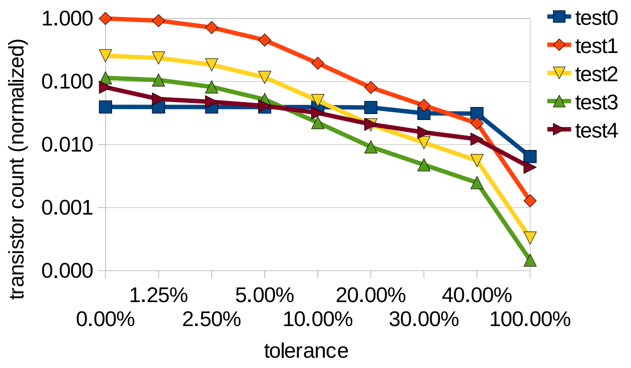


Fig. 3. The number of transistors in the hierarchy recovered netlists as a function of the parameter tolerance. The transistor sizes are normalized to the number of transistors in the "pushed" netlist; thus a transistor number of 1.0 has the same number of transistors as the "pushed" netlist.

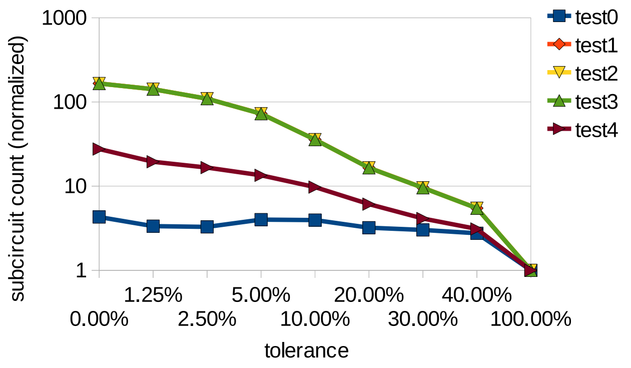


Fig. 4. The number of subcircuits in the hierarchy recovered netlists as a function of the parameter tolerance. The number of subcircuits is normalized to the number of subcircuits in the "pushed" netlist; thus a subcircuit number of 1.0 has the same number of subcircuits as the "pushed" netlist. Note that the hierarchy recovery works by creating type-variants of each subcircuit to account for differences in the parameters. Thus, the number of subcircuits is always greater than the original netlist.

B. *Experiment 2: Standard Cell Recovery*

Another example application of post-layout hierarchy recovery comes from recovering the standard cells from the extracted layout of a lithography simulated design. The idea is to capture the systematic (e.g. predictable) variations in the transistor geometries using type-variants of each cell. The resulting netlist can then be used to improve power and timing estimates.

TABLE II
NETLIST VERIFICATION TIMES

	hier. recovery runtime (s)		LVS runtime (s)			ERC runtimes (s)		
	$R_{0\%}$	$R_{10\%}$	pushed	$R_{0\%}$	$R_{10\%}$	pushed	$R_{0\%}$	$R_{10\%}$
test0	72	58	179	15	13	–	–	–
test1	7	2	9	5	5	158	141	69
test2	12	6	39	6	5	568	545	190
test3	28	18	114	6	6	1302	724	369
test4	15	10	6	5	5	160	81	65

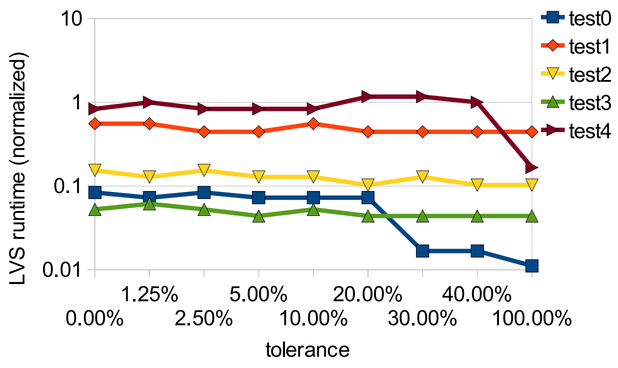


Fig. 5. The runtime of LVS netlist comparisons as a function of the parameter tolerance. The runtimes are given as a fraction of the "pushed" netlist runtime, thus a runtime of 1.0 is equal to the runtime of the "pushed" netlist. The runtime of the test4 fluctuates, but are only +/- 1s from $R_{0\%}$ to $R_{40\%}$ and within rounding error.

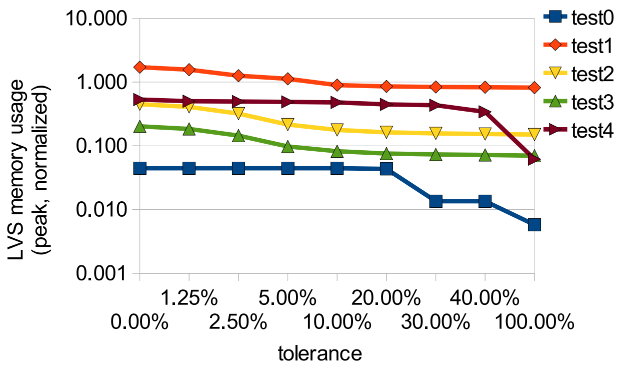


Fig. 6. The peak memory usage during the LVS netlist comparisons as a function of the parameter tolerance. The memory usage is given as a fraction of the "pushed" netlist memory, thus a memory usage of 1.0 is equal to the memory usage of the "pushed" netlist.

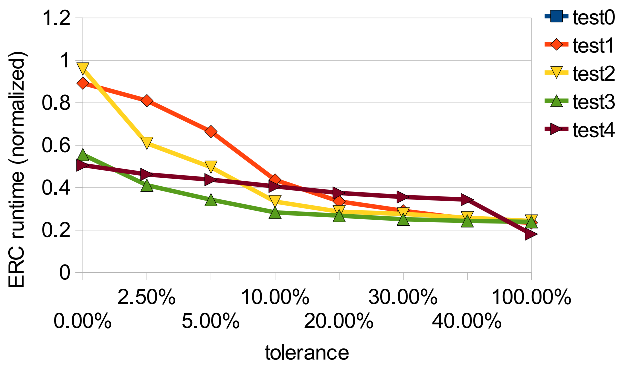


Fig. 7. The runtime of electrical rule checking as a function of the parameter tolerance. The runtimes are given as a fraction of the "pushed" netlist electrical rule check time, thus a runtime of 1.0 is equal to the runtime of the "pushed" rule check. Note that the Y-axis is on an absolute scale, and not on a logarithmic scale.

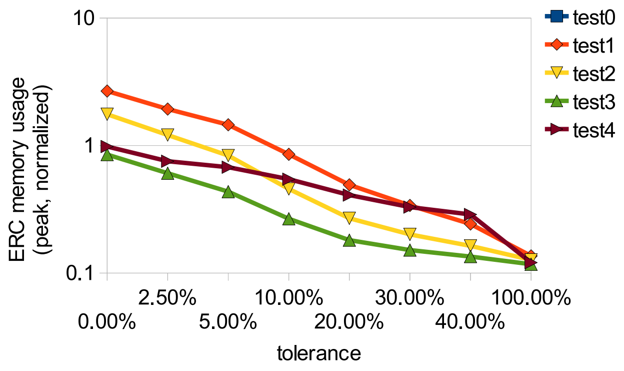


Fig. 8. The peak memory usage of electrical rule checking as a function of the parameter tolerance. The memory usage is given as a fraction of the "pushed" netlist electrical rule check memory usage, thus a usage of 1.0 is equal to the memory usage of the "pushed" rule check.

The idea of lithography extracted timing is discussed in [7], [8], [9], [10]. In the normal timing sign-off flow, the netlist is timed with with the assumption that the lithography is ideal, and the gates printed as drawn. However there is an advantage that can be gained by utilizing the extra information into the design flow, and the papers above consider using this information to adjust the timing estimates. However, they do not consider the idea of type-variants[6].

In this experiment, the layouts, with the extracted lengths and widths, are run through the hierarchy recovery algorithm for three ISCAS '89 benchmarks and an arithmetic logic unit from OpenCores.org [11]. The benchmarks, and corresponding information about the systematic variation due to lithography is shown in Table III. In the table, the variations under nominal lithography conditions have approximately zero mean, and standard deviations that are 3nm in gate width, and 2nm in gate length. However at non-ideal lithography conditions, the standard deviations in l can double, and cause a substantial shift in the mean.

A table of the results is shown in Table IV. The N=k

[6][7] does, however, use predefined variants.

TABLE III
EXPERIMENT 2- SYSTEMATIC w AND l VARIATIONS

	Nominal Lithography				Exposure .9 / Defocus 80nm			
	$\mu_{\text{error}}(w)$	$\sigma_{\text{error}}(w)$	$\mu_{\text{error}}(l)$	$\sigma_{\text{error}}(l)$	$\mu_{\text{error}}(w)$	$\sigma_{\text{error}}(w)$	$\mu_{\text{error}}(l)$	$\sigma_{\text{error}}(l)$
	[nm], (Gate length is 50nm)							
s35932	1.34	3.39	-.54	2.06	.54	4.33	3.01	2.37
s38417	-1.03	3.32	.11	1.92	.24	4.61	3.69	2.17
s38584	1.16	3.75	.15	1.86	.30	5.20	3.30	2.27
alu	1.09	4.91	.23	1.53	-.15	6.91	3.54	1.94

	Exposure .8 / Defocus 160nm			
	$\mu_{\text{error}}(w)$	$\sigma_{\text{error}}(w)$	$\mu_{\text{error}}(l)$	$\sigma_{\text{error}}(l)$
	[nm], (Gate length is 50nm)			
s35932	-2.4	7.93	4.83	4.44
s38417	-2.85	1.02	5.56	4.36
s38584	-2.81	9.03	4.84	4.70
alu	-3.7	10.51	4.74	4.50

TABLE IV
EXPERIMENT 2- GATE LENGTH STATISTICS AFTER TYPE-MAPPING

	Nominal		N=1		N=2		$\tau = 10\%$			$\tau = 5\%$		
	σ_l	$\Delta_{l,\max}$	σ_l	$\Delta_{l,\max}$	σ_l	$\Delta_{l,\max}$	σ_l	$\Delta_{l,\max}$	$\#_{\text{variants}}$	σ_l	$\Delta_{l,\max}$	$\#_{\text{variants}}$
Exposure .9 / Defocus 80nm												
s35932	2.34	10.2	1.02	9.3	.46	9.3	.54	2.5	31	.42	1.2	65
s38417	2.17	10.3	2.54	10.3	.68	9.6	.53	2.5	63	.42	1.2	111
s38584	2.28	10.3	.93	10.1	.66	10.1	.57	2.5	66	.42	1.2	123
alu	1.92	8.3	1.21	8.1	.87	8.0	.68	2.5	60	.40	1.2	95

Exposure .8 / Defocus 160nm												
s35932	4.43	16.6	1.30	19.8	.71	19.7	.57	2.5	64	.48	1.2	206
s38417	4.37	16.2	4.28	16.2	1.13	15.9	.63	2.5	109	.49	1.2	339
s38584	4.68	15.4	3.5	15.4	1.1	15.8	.65	2.5	114	.49	1.2	360
alu	4.50	14.6	1.89	14.6	1.64	14.6	.75	2.5	84	.49	1.2	219

columns refer to clustering using the method in Section III-B1 with k clusters, and in these cases, the nominal cell is also available (thus, "N=0" is equivalent to the nominal case). The "τ=x%" columns refer to clustering using the method in Section III-B2 using a tolerance set to $x\%$ of the nominal values. For example, a nominal gate length of 50nm with a tolerance of 2% is equivalent to a 1nm tolerance. In both methods, the standard deviation of the errors decreases quickly as the number of clusters increases, and as the tolerance decreases. However, the maximum error in gate length $\Delta_{l,\max}$ does not necessarily decrease as the number of clusters increases. This indicates that the tolerance metric (4) may be a more reliable measure when the maximum deviations are important.

It is interesting to note that the changes in lithography may cause non-zero mean errors. For example, Table III shows that the gate lengths tend to be larger when the lithography parameters are degraded. For the nominal case, the mean gate length error is approximately 0nm, while for Exposure .9 / Defocus 80nm, the mean length error is +3.7nm, and for Exposure .8 / Defocus 160nm, the mean length error is +5.7nm.

When type-variant creation is performed, the standard deviation in the length drops dramatically from approximately 2nm to approximately 0.5nm. This is because the systematic variation from the lithography affects each transistor in a similar way, and the variations can be reduced using recentering the dimensions of each transistor. For example, in the s35932 circuit, the NAND2X2 gates are recentered to the dimensions in Table V. Each transistor differs from the drawn dimensions in a systematic way, and thus a large amount of information can be recovered by adjusting the mean. However, this does not imply that all M0 transistors in the NAND2X2 have the same gate length; the histogram is shown in Figure 9.

TABLE V
NAND2X2 VARIATION IN S35932

	w(drawn)	w(mean)	l(drawn)	l(mean)
	[nm]			
M0	270	261.1	50	52.4
M1	270	260.9	50	53.2
M2	260	255.0	50	54.3
M3	260	252.6	50	53.9

The new type-variants can be used to improve timing estimates. For example, suppose that a timing accuracy of 1%

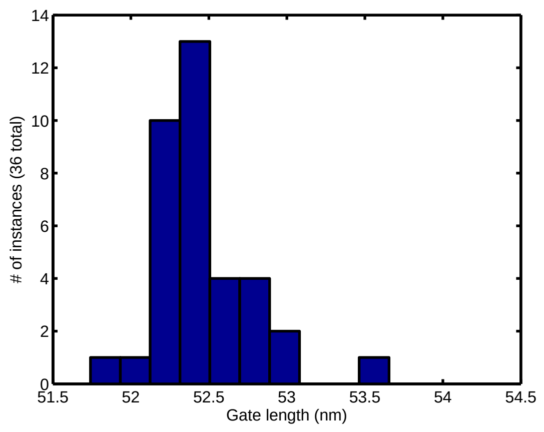
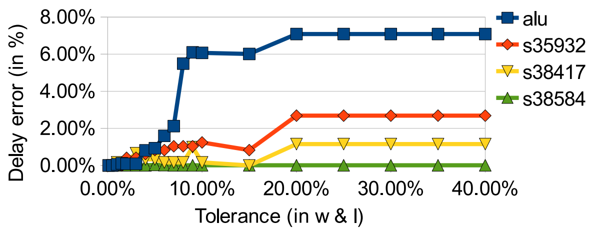
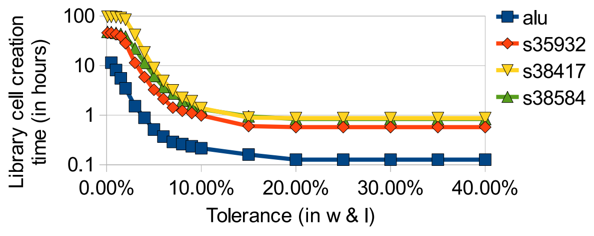


Fig. 9. Histogram of the gate lengths of the NAND2X2 transistor M0 in s35932. The mean is 52.7nm, which differs from the drawn length of 50nm.



(a)



(b)

Fig. 10. The top graph plots the errors in delay as a function of the tolerance τ for the benchmarks in Experiment 2. In these examples, a tolerance of $< 10\%$ gives an error less than 2%. The bottom graph shows the estimated library characterization time as a function of the tolerance τ for the benchmarks in Experiment 2. The runtime for small tolerances is very high (50-100 hours).

is needed. Using the approximation

$$\text{Delay} \propto \frac{C}{I} \approx \frac{L}{W}\frac{C}{(V_{\text{GS}} - V_{\text{T}})^2}, \tag{5}$$

this roughly translates to tolerance of $\tau = .5\%$ in length and widths. This tolerance is measured with respect to the nominal values in gate length and gate width, as opposed to the parametric spread used in Section IV-A.

Experiments are run to measure the effect of the tolerance τ on the circuit delay for the lithography condition Exposure .8 / Defocus 160nm. The top plot in Figure 10 plots the error in the timing as a function of the tolerance τ. The delays are calculated using transistor level netlists that are created by the hierarchy recovery program, using the commercial tool NanoTime [12]. The transistor widths and lengths are extracted as the maximum width, and the average gate length of the simulated lithography printing, respectively[7]. In these cases, a tolerance of less than 10% is needed to achieve a delay accuracy of 2%. To achieve an accuracy of 1% however, the tolerance needs to decrease to 3%.

The number of extra cells that are needed to satisfy the tolerance is prohibitively high. For example, using a tolerance of 3% on the s38584 circuit requires 651 extra cells (i.e. type-variants), which would require approximately 17 hours to run in the library characterization tool Liberty NCX [14][8]. A plot of the runtimes vs. the tolerance is shown at the bottom of Figure 10.

Fortunately, there is a way to reduce the runtimes. This is because the number of gates that determine the maximum delay of the circuit is much smaller than the total number of gates. Thus, initial slack estimates can be used to adjust the tolerances on each of the gates, giving non-critical cells a larger tolerance than the cells that are non-critical. More formally, if gate i has the corresponding minimum slack path π_i that runs through it, the tolerance of each gate is adjusted as:

$$\tau_i = \min_{\pi_i} \left\{ \left(\frac{\text{slack}(\pi_i)}{\alpha \cdot \text{delay}(\pi_i)} + \tau \right), \tau_{\max} \right\} \tag{6}$$

where τ is the desired tolerance, τ_i is the adjusted tolerance for the widths and lengths of the gate, and $\text{slack}(\pi_i)$ and $\text{delay}(\pi_i)$ are the slack and the delay of the paths, respectively. α is a term that corrects for modeling errors and is set to $\alpha = 2.0$, and $\tau_{\max}$ is the maximum allowed error, and is set to 20%.

Intuitively, the the ratio

$$\frac{\text{slack}(\pi_i)}{\alpha \cdot \text{delay}(\pi_i)} \tag{7}$$

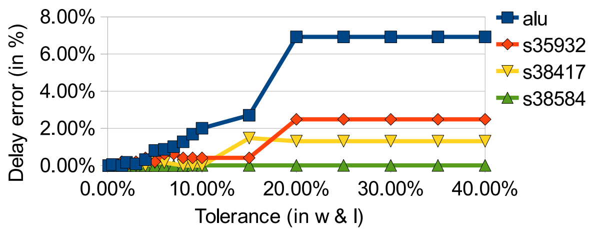
is related to the percentage change in the delay before the gate becomes critical. Thus, if a gate has a slack of .3ns, and a corresponding path delay of $.6ns$, then the delay of each gate in that path can vary by 50% before it becomes critical. Dividing this by α accounts for modeling errors, and this is added to the original tolerance for this gate. In other words, it is assumed that a change less than 25% in the gate parameters will not affect the worst-case delay.

The results for the slack-adjusted case is shown in Figure 11. The error vs the tolerance is similar to the case where no slack is used, showing that the critical cells are accounted for appropriately. However, the library characterization runtimes are dramatically smaller (approximately 1 hour and less, compared to 10+ hours), as the number of type-variant cells that are needed is decreased.

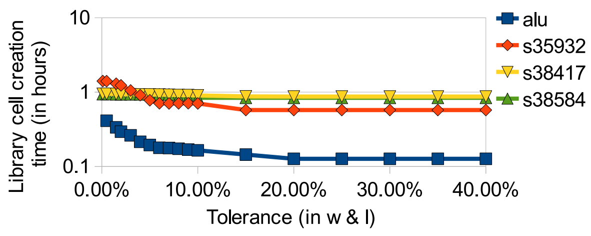
A similar experiment can be run to improve the circuit power estimates in the Exposure .8 / Defocus 160nm process point. In this case, adding one extra type-variant (e.g. N=1) can correct to approximately 2% of the correct values. For example, in the alu circuit, the nominal power values have a -4% error in total power and a -11% error in leakage

[7]Note that better models, such as those in [10] can be used to improve the accuracy of the effective transistor dimensions.

[8]The runtimes are estimated by adding the time it takes to characterize the nominal versions of each cell. As a rough guide, the time per cell, in seconds, is roughly proportional to $[\#\ \text{inputs}]^{.85}[\#\ \text{outputs}]^{2.1}[\#\ \text{transistors}]^{.62}$.



(a)



(b)

Fig. 11. The top graph plots the errors in delay as a function of the tolerance τ for the benchmarks in Experiment 2, when the slacks are available to adjust the tolerances. In these examples, a tolerance of $< 10\%$ gives an error less than 2%. The bottom graph shows the estimated library characterization time as a function of the tolerance τ for the benchmarks in Experiment 2. The runtimes are much smaller than those in Figure 10 .

power[9]. Adding one extra type-variant reduces the error to $-.4\%$ in total power and -1.2% in leakage power, and costs only 6 minutes in library characterization time. In the s38417 circuit, the nominal power values have a -5% error in total power and a -8.5% error in leakage power. Adding one extra type-variant reduces the error to approximately -1% in total power and -1% in leakage power, and costs only 19 minutes in library characterization time.

V. Summary

In this paper, we presented a method to recover the hierarchy in layout extracted netlists. The idea is to account for variations in each type by creating type-variants, and using these variants to recover the hierarchy. Applications in validation show that this can result in a significant improvement in runtime, and a reduction in the size of the netlist. Furthermore, applications in timing and power show that this method can improve the accuracy of timing and power estimates with a small library characterization overhead. The experiments show that in physical verification, this method leads to a 70% reduction in runtime on average, without any parametric error; furthermore tractable timing and power analysis can be performed that utilizes detailed transistor information. In the future, the work can be extended to hierarchy degradation coming from wire parasitics as well.

VI. Acknowledgments

We would like to thank Dr. Saumil Shah and Amarnath Kasibhatla for some early discussions and experiments.

[9]These experiments were run using Liberty NCX [14] to characterize the library, and Encounter [15] to compute the power of the design.

References

[1] T. Lengauer and K. Wagner, “The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems,” *Journal of Computer and System Sciences*, vol. 44, no. 1, pp. 63–93, 1992.

[2] M. Igusa, H. Chen, S. Chao, W. Dai, and D. Shyong, “Design hierarchy-based placement,” Jun. 19 2001, uS Patent 6,249,902.

[3] P. Russell and G. Weinert, “System and method for verifying a hierarchical circuit design,” Jun. 18 1996, uS Patent 5,528,508.

[4] A. Jain, M. Murty, and P. Flynn, “Data clustering: a review,” *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[5] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 281-297. California, USA, 1967, p. 14.

[6] Mentor Graphics, “Calibre v2010.2_38.23,” http://www.mentor.com/, 2010.

[7] P. Gupta and F. Heng, “Toward a systematic-variation aware timing methodology,” in *Proceedings of the 41st annual Design Automation Conference*. ACM, 2004, p. 326.

[8] J. Yang, L. Capodieci, and D. Sylvester, “Advanced timing analysis based on post-OPC extraction of critical dimensions,” in *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005, pp. 359–364.

[9] P. Gupta, A. Kahng, S. Nakagawa, S. Shah, and P. Sharma, “Lithography simulation-based full-chip design analyses,” in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 6156, 2006, pp. 277–284.

[10] T. Chan, R. Ghaida, and P. Gupta, “Electrical Modeling of Lithographic Imperfections,” in *2010 23rd International Conference on VLSI Design*. IEEE, 2010, pp. 423–428.

[11] Available from http://www.opencores.org.

[12] Synopsys , “Nanotime a-2007.12-sp1,” http://www.synopsys.com/, 2008.

[13] T. Chan, A. Kagalwalla, and P. Gupta, “Measurement and optimization of electrical process window,” in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 7641, 2010, p. 15.

[14] Synopsys, “Liberty ncx d-2009.12-sp3,” http://www.synopsys.com/, 2010.

[15] Cadence, “Soc encounter 6.2,” http://www.cadence.com/, 2007.