Parametric Hierarchy Recovery in Layout Extracted Netlists

John Lee and Puneet Gupta
Electrical Engineering
UCLA
lee@ee.ucla.edu, puneet@ee.ucla.edu

Fedor Pikus
Formerly at Mentor Graphics
fedor_pikus@mentor.com

***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, extraction tools flatten these designs. As a result, in post-layout extraction, the hierarchy is lost and the designs are flattened, increasing 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 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 of 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 is 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. 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.

This causes the following problems:

- The resulting netlist is much larger than the hierarchical netlist.
- It is difficult to debug errors in verification, as millions of errors in the flattened netlist may be due an error in one block in the hierarchy.
- Slower runtimes in downstream tools. The removal of hierarchy makes the netlists larger, and slower to process.

While the effective dimensions of this context may be much larger than the size of a cell, the repetitive nature of modern designs imply that the number of contexts is usually fairly limited. While devices in different placements of the same cell have different parameters, these differences can be represented by introducing multiple variants of the original cell, each with different parameters.

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 recovered hierarchy helps to manage the complexity, and enable faster runtimes. The contributions of this paper are as follows.

- We give algorithms for recovering hierarchy, in the context of verification, and post-layout timing.
- Results that show that hierarchy recovery is effective at reducing runtime and netlist size, and provides tractable timing and power analysis.

Section II describes the methods used to recover hierarchy. Section IV-A presents applications of this method in verification, and Section IV-B presents applications of this method in standard cell recovery. The paper is concluded in Section V.

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 to 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*.

III. Methods for Type Mapping and Type Creation

Type mapping and type creation are very similar to the *data 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.

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

$$\text{Error}_{\text{tol}}(\mathcal{I}) = \begin{cases} \infty & \text{if } \max_i\{|p_i - p_{\text{type}_{\text{variant}}(i)}|\} > p_{\text{tol}} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

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 the type-mapping application of this algorithm, the number of type-variants for each type is *a given input*, and the assignment to the clusters is refined iteratively in two steps. In the first step, each instance is assigned to the type-variant that minimizes the error. Next, the parameters of each type-variant is updated to minimize the error.

2) Type-mapping with tolerance: The k-Means algorithm cannot be used effectively when a tolerance-based type-mapping is needed 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 type-mapping is performed using a heuristic that assigns each instance to the first matching type-variant. If there are remaining unmatched instances, a new type-variant is formed by using a modified version of the k-means++ algorithm [6][2]. In this method, the unmapped instance that is farthest, with the largest parametric difference deviation (in a $||\cdot||_\infty$ sense), from all current type-variants is assigned to the new type-variant. This has the interpretation of choosing the furthest outlier as the new type-variant. The unmapped instances are then checked to see if they can be mapped with the new type-variant. This process is iterated until no remaining members are left.

At the beginning of the method, when no type-variants have been created, the choice of which instance that should be chosen first affects the performance of the algorithm. Thus, along the lines of [6], we perform the type-mapping multiple times, each with a different initial instance, and the type-mapping with the least number of type-variants is used. In the experiments in Section IV-B, the clustering is performed five times, and provides a 2% to 12.8% reduction in type-variants, with an average reduction of 7.8%. The method is summarized as follows:

1) **Start:** All instances $\mathcal{I}$ unclustered, and empty set of type-variants ($\mathcal{T}_{\text{variant}} = \{\}$)
2) Randomly assign an instance to the first type-variant
3) **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. $\text{Error}_{\text{tol}} = 0$). If none are satisfied, then the instance is left unmapped
4) If there are unmapped instances remaining, assign the instance with the largest parametric distance ($||p - p_{\text{type}_{\text{variant}}(k)}||_\infty$) over all k to a new type-variant. Add

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

[2] The k-means++ algorithm works for norm-based clustering. We interpret tolerance based mapping as similar to clustering with the $||\cdot||_\infty$ norm, and this is used to adapt the k-means++ algorithm.

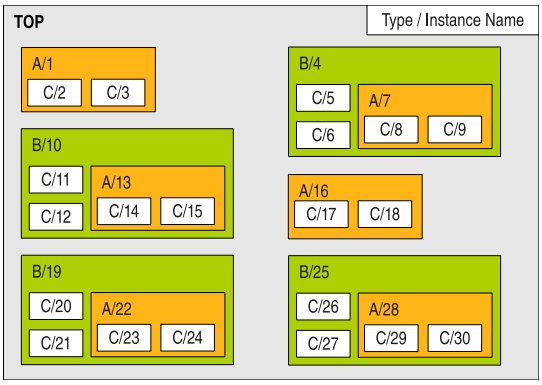

Fig. 1. Example of a hierarchical design. The design currently has types A, B, and C with 30 instances total. Instances in {1, 7, 13, 16, 22, 28} are of type A; instances in {4, 10, 19, 25} are of type B; the remainder are of type C.

the type-variant to $\mathcal{T}_{\text{variant}}$, and repeat Step 3 above. Otherwise exit.

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. In this example, the higher-level mapping of the top blocks, A and B, *induces*, or forces, the matching instances at the lower-levels to be clustered together. These clusters must be maintained when the lower-level instances are clustered together. For example, if B/4 and B/10 are mapped to the same type-variant, then A/7 and A/13 must also be mapped together, as well as C/8 and C/14, among others.

To perform hierarchical type mapping, while accounting for the interactions between different levels of hierarchy, the following method is used:

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

This provides a top-down approach for clustering. This method uses the hierarchy in the design, and the concept of induced clusters to provide a straightforward method for hierarchical clustering. By starting with the top level, and working down, the complex interactions between hierarchies can be accounted for. In contrast to Section III-B2, this algorithm does not have a random part to it. Due to the interactions of the higher levels on the clustering of lower level instances, the improvements gained by using a variation of the k-means++ approach is $< 1\%$.

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 the transistor information is extracted from the layout, along with the connectivity information between transistors. Next, the extracted information is compared against the original netlist to verify that the two netlists are the same.

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

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, where devices that are identical are "pushed" down the hierarchy. 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 transistor cannot be "pushed" down the hierarchy, 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.

TABLE I
EXPERIMENT 1- BENCHMARKS

	flat	pushed		$R_{0\%}$	
	# trans	# trans	# subckt	# trans	# subckt
test0	17,505,258	2,330,409	219	92,236	941
test1	834,062	834,062	368	833,456	61,108
test2	3,336,248	3,256,313	369	833,468	61,112
test3	7,506,558	7,293,398	369	833,468	61,112
test4	2,354,107	1,027,717	1219	83,405	33,623

TABLE II
NETLIST VERIFICATION TIMES

Runtimes (s)					
	Typemap	LVS		ERC	
	$R_{0\%}$	pushed	$R_{0\%}$	pushed	$R_{0\%}$
test0	72	179 (1.0)	15 (.08)	–	–
test1	7	9 (1.0)	5 (.56)	158 (1.0)	141 (.89)
test2	12	39 (1.0)	6 (.15)	568 (1.0)	545 (.96)
test3	28	114 (1.0)	6 (.05)	1302 (1.0)	724 (.56)
test4	15	6 (1.0)	5 (.83)	160 (1.0)	81 (.51)

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

```
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 multi-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 range of tolerances, $R_{X\%}$, where X is the percentage tolerance with respect to the maximum deviation of the parameter. Thus, for example, if the well proximity effect parameter "sca" has a *maximum deviation* between instances of 9.8, then a $R_{10\%}$ would set the tolerance to be 0.98. The sizes of these circuits are summarized in Table I which lists the number of transistors in the total design, the pushed netlist, and the recovered netlist with zero tolerance ($R_{0\%}$).

Table I shows that the number of transistors in the recovered netlist with zero tolerance is generally much 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 2(a) and 2(b) summarize the transistor count and the subcircuit count, as a function of the tolerance.

We measure the effect of hierarchy recovery on the Layout vs. Schematic (LVS) comparison process using Calibre [7]. In this experiment, 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 subcircuits, 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 subcircuits 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[3] is smaller 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 slowest comparisons (test0 and test3) benefit the most, and are faster with the hierarchy recovery. A plot of the runtime vs. tolerance is shown in Figure 2(c).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 Figure 2(d) show an example of an ERC run in Calibre [7] for a set of common checks[4]. 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 significant improvement in runtime that increases as the tolerance increases.

[3]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.

[4]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.

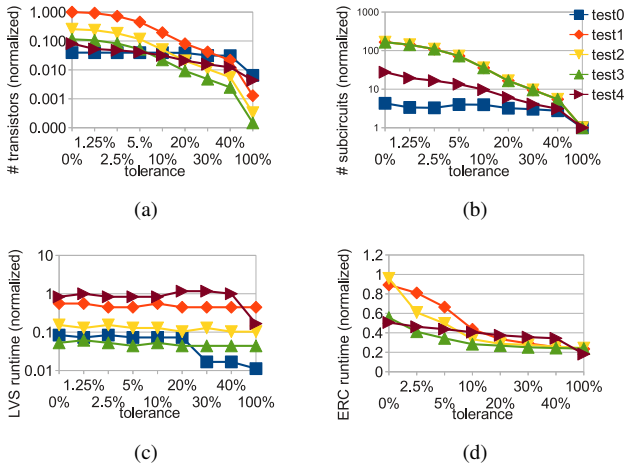

Fig. 2. Performance plots for hierarchy recovery. All plots are normalized to the "pushed" netlist; thus a value of 1.0 is equivalent to the "pushed" netlist. (a) gives transistor counts; (b) gives the subcircuit (type-variant) counts; (c) gives the LVS runtime; and (d) gives the ERC runtime.

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. These variations come from imperfections in the lithography conditions, such as defocus and exposure. The resulting netlist can then be used to improve power and timing estimates.

The idea of lithographically extracted timing is discussed in [8], [9], [10], [11]. In the normal timing sign-off flow, the netlist is timed 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[5].

In this paper, three lithography process conditions: Nominal (Exposure= 1.0 / Defocus= 0nm); Exposure= .9 / Defocus= 80nm; and Exposure= .8 / Defocus= 160nm are considered. Under nominal lithography conditions, the width and lengths 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. For example, while in the nominal case, the mean gate length error is approximately 0nm, in the Exposure .9 / Defocus 80nm case, the mean length error is +3.7nm, and for the Exposure .8 / Defocus 160nm case, the mean length error is +5.7nm.

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 [12].

The new type-variants can be used to improve timing estimates. For example, suppose that a timing accuracy of 1% is needed. The approximate model $\text{Delay} \propto \frac{C}{I} \approx \frac{L}{W}\frac{C}{(V_{GS}-V_T)^2}$, can be used to type-mapping for timing. Thus, instances that have their L/W values within a tolerance will be mapped to the same type-variant.[6]

[5][8] does, however, use predefined variants.

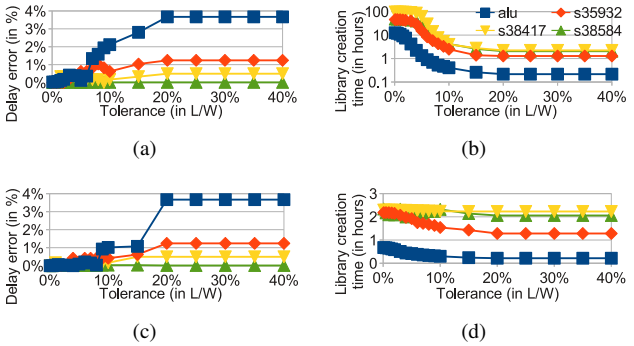

Fig. 3. Hierarchy recovery applied to post-layout timing. The top plots show timing results when a strict tolerance is used for type-mapping while the bottom plots show the results when slacks are used to adjust the tolerance. The left-side plots show the delay error vs. tolerance, and right-side plots shows the library characterization time vs. the tolerance. The delay error in (c) is similar to (a), but the corresponding library characterization time in (d) is much smaller than (b).

Experiments are run to measure the effect of the tolerance τ on the circuit delay for the lithography condition Exposure .8 / Defocus 160nm using the model above. The tolerance is a function of the L/W for each transistor. The top plot in Figure 3 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 [13]. 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 7%.

A tolerance-based clustering is used because timing applications are sensitive to outliers. For example, with a square-error, while the total error may be small, there may be outliers – sporadic instances with a large square-error. If the outlier falls on the critical path, this will cause the timing-estimates to be unreliable. Thus, for reliable timing, a tolerance is used to guarantee the reliability of the timing estimates.

The number of extra cells that are needed to satisfy the tolerance may be prohibitively high. For example, using a tolerance of 7% on the s38417 circuit requires 474 extra cells (i.e. type-variants), which would require approximately 14 hours to run in the library characterization tool Liberty NCX [14][8]. Plots of the delay error vs. tolerance and the runtimes vs. the tolerance are shown in Figures 3(a) and 3(b).

[6]Note more sophisticated models are also possible when additional parametric variation is present. For example, models that incorporate the effects of stress, doping and line-edge roughness can be used.

[7]Note that better models, such as those in [11] 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 $[\#\text{ inputs}]^{.85}[\#\text{ outputs}]^{2.1}[\#\text{ transistors}]^{.62}$.

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 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:

$$\tau_i = \min_{\pi_i} \left\{ \left(\frac{\text{slack}(\pi_i)}{\alpha \cdot \text{delay}(\pi_i)} + \tau \right), \tau_{\max} \right\} \quad (3)$$

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 ratio $\frac{\text{slack}(\pi_i)}{\alpha \cdot \text{delay}(\pi_i)}$ 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 this case, it can be assumed that a change that is less than 25% in the gate parameters will not affect the worst-case delay.

The results for the slack-adjusted case is shown in Figures 3(c) and 3(d). 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 2.5 hours and under, compared to 10+ hours), as the number of type-variant cells that are needed is decreased[9].

The library characterization runtime can also be decreased by reusing previous library characterizations. For a tolerance $\tau = 4\%$, the runtime reductions are between 1% and 22%, with an average of 7%, when the library created from a different benchmark is reused. This suggests that there is significant overlap between designs and that a library of prior characterizations can be reused in the future to reduce future library characterization runtimes.

In the case of library characterization for power analysis, re-centering the transistor dimensions provides adequate accuracy – adding one type-variant (e.g. N=1) that contains the mean parameters can correct to approximately 2% of the correct values. For example, in the alu circuit, while the nominal power values have a -4% error in total power and a -11% error in leakage power[10], 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, while 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. This translates into only 19 minutes in library characterization time. Thus, accurate power estimates can be performed with little cost.

[9]Note that the runtime of the type-mapping itself is less than a minute for all benchmarks and tolerances, and is thus negligible in comparison to the library characterization time.

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

V. Summary

In this paper, we presented a method to recover the hierarchy in layout extracted netlists. The idea is to account for variations between instances of a 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. In the case of post-layout timing, slack-weighted type-mapping can be used to reduce the library characterization needed from over 12 hours to under 2.5 hours. 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.

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 *Proc. of Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 281-297. , 1967, p. 14.

[6] D. Arthur and S. Vassilvitskii, "k-means++: the advantages of careful seeding," in *Proc. ACM-SIAM Symposium on Discrete algorithms*, 2007, pp. 1027–1035.

[7] Mentor Graphics, "Calibre v2010.2_38.23," http://www.mentor.com/, 2010.

[8] P. Gupta and F. Heng, "Toward a systematic-variation aware timing methodology," in *Proc. Design Automation Conference*. , 2004, p. 326.

[9] J. Yang, L. Capodieci, and D. Sylvester, "Advanced timing analysis based on post-OPC extraction of critical dimensions," in *Proc. Design Automation Conference*. ACM, 2005, pp. 359–364.

[10] 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.

[11] T. Chan, R. Ghaida, and P. Gupta, "Electrical Modeling of Lithographic Imperfections," in *International Conference on VLSI Design*. , 2010, pp. 423–428.

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

[13] Synopsys , "Nanotime a-2007.12-sp1," http://www.synopsys.com/, 2008.

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

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