ABSTRACT
As designs continue to grow in size and complexity, EDA paradigm shifts from flat to hierarchical timing analysis. In this paper, we propose compact and accurate timing macro modeling, which is the key to achieve efficient and accurate hierarchical timing analysis. Our macro model tries to contain only a minimal amount of interface logic. For timing graph reduction, we propose anchor pin insertion and deletion by generalizing existing reduction techniques. Furthermore, we devise a lookup table index selection technique to achieve high model accuracy over the possible operating condition range. Compared with two common models used in industry, extracted timing model and interface logic model, our model has high model accuracy and small model size. Based on the TAU 2016 timing contest on macro modeling benchmark suite, our results show that our algorithm delivers superior efficiency and accuracy: Hierarchical timing analysis using our model can significantly reduce runtime and memory compared with flat timing analysis on the original design. Moreover, our algorithm outperforms TAU 2016 contest winner in model accuracy, model size, model usage runtime and memory.

Keywords
Static timing analysis; hierarchical timing analysis; timing macro modeling; extracted timing model; interface logic model.

1. INTRODUCTION
As design evolution continues, designs rapidly grow in size and complexity. IP reuse and hierarchical design are the key to bridge the design productivity gap. A large-scale integration design can be hierarchically partitioned into manageable blocks that can be implemented in parallel (many of them are duplicate IPs), thus saving design time.

During chip design, static timing analysis (STA) is recognized as an essential task that directly influences design cycle time. To handle a billion gate design, traditional full-chip flat timing analysis may take days and consume much memory. Therefore, along with the hierarchical design trend, EDA paradigm shifts from flat to hierarchical timing analysis to ease the time-to-market pressure [1][2][3].

Hierarchical timing analysis is enabled by analyzing a block once, generating the corresponding simplified timing macro model, and reusing the macro model for all duplicate blocks at the parent or top level of hierarchy to alleviate detailed analysis. Efficient and accurate hierarchical timing analysis relies on compact and accurate macro modeling. Nevertheless, there is a tradeoff among model size, model accuracy, and model generation time.

In addition to the tradeoff, the main challenge of timing macro modeling is context independency [4]. Context means the conditions under which a block is analyzed. To avoid accuracy loss, the context range considered during model generation ideally should cover the whole scope within which the block will be operated at upper-level STA. One example of reflecting the difficulty of context independency is the clock common path pessimism removal (CPPR) effect. It is difficult to model the CPPR credits of the top-level common source at the block-level analysis.

Two most commonly used timing macro models in physical design are the extracted timing model (ETM) and the interface logic model (ILM) [5]. ETM contains timing arcs for only block
interfaces and frequently takes the form of a Liberty model [6] (see Fig. 1(b)). ETM is an abstract model hiding implementation details; it is suitable for IP providers, but incurs some accuracy loss and cannot handle the CPPR effect. ILM is a partial gate-level netlist containing only interface logic (see Fig. 1(c)). ILM is more accurate than ETM, but larger. On the other hand, to reduce the timing graph size, several graph reduction techniques have been investigated in literature. For example, serial and parallel merging is developed in [7]; biclique-star replacement is proposed in [8]; tree merging is devised in [9].

In this paper, we propose a novel algorithm to generate a compact and accurate timing macro model. For achieving context independency, efficiency, and accuracy, our key idea is to retain only a small amount of necessary interface logic (removing all unnecessary and unobservable logic, while preserving every pin that may affect accuracy). In our algorithm flow, we first convert a design into an initial timing graph. Second, we capture interface logic and remove internal logic. Third, we mark necessary pins, including pins on gates in the first few stages within the fanout cone of each input port, on gates directly connected to output ports, or on gates with multiple fanouts in the clock network. Fourth, we iteratively reduce the timing graph without touching the marked pins. We devise anchor pin insertion and deletion by generalizing existing graph reduction techniques. Combining with parallel merging and clock network merging, our reduction technique leads to a small model size and guarantees to recover CPPR credits. Fifth, if a restricted Liberty model is used (e.g., [10]), we additionally insert pseudo pins. Finally, to achieve high accuracy over the possible operating condition range, we determine proper lookup table indices for timing arcs and output a timing macro model. Compared with ETM and ILM, our model has high accuracy and small model size. (see Fig. 1 (d).)

Our experiments are conducted on the TAU 2016 timing contest on macro modeling benchmark suite [10]. Experimental results show that our algorithm delivers superior efficiency and accuracy: Hierarchical timing analysis using our model can significantly reduce runtime and memory compared with flat timing analysis on the original design. Moreover, our algorithm outperforms TAU 2016 contest winner in model accuracy, model size, model usage runtime and memory.

The remainder of this paper is organized as follows. Section 2 briefly introduces ETM, ILM, and problem formulation. Section 3 presents our timing macro model generation algorithm. Section 4 describes our lookup table index selection. Section 5 shows experimental results. Finally, Section 6 concludes this work and indicates future directions.

2. PRELIMINARIES AND PROBLEM FORMULATION

In this section, we briefly introduce two common timing macro models and give the problem formulation.

2.1 Extracted Timing Model (ETM)

An ETM abstracts the interface behavior of a block and replaces the original netlist with a library cell (frequently in Liberty format) (see Fig. 1(b)). The cell contains only pin-to-pin timing arcs, and each arc corresponds to a timing path in the original design from an input port to a register, from an input port to an output port, and from a register to an output port. Generally, an ETM has long model generation time but very short runtime when it is used at top-level timing analysis. Because this model provides a reasonable accuracy and hides detailed design implementation information, it is ideal for IP providers.

2.2 Interface Logic Model (ILM)

An ILM is a partial gate-level netlist that contains only the interface logic of a block and eliminates the internal register-to-register logic (see Fig. 1(c)). An ILM includes the combinational logic from each input port to the first stage of registers of the block, the combinational logic from the last stage of registers to each output port of the block, the clock paths to these registers, and combinational paths from the input ports that do not encounter a register and pass directly to output ports. Generally, an ILM maintains a good balance among model size, model accuracy, and model generation time.

2.3 Problem Formulation

As mentioned earlier, compact and accurate timing macro modeling is the key to achieve efficient and accurate hierarchical timing analysis. In this paper, we adopt the timing macro modeling problem formulation used in the TAU 2016 timing contest on macro modeling, which can be described as follows.

The timing macro modeling problem: Given the gate-level netlist and the net parasitics of a block, the early and late cell libraries (describing cell functionality, early and late timing information), and timing assertions (operating conditions), the goal is to create a timing macro model that encapsulates the timing behavior of the block such that the model size is minimized and accuracy satisfies a given error bound.

To satisfy the accuracy requirement, the generated macro model shall include sufficient timing arcs and internal pins. The accuracy is measured by comparing hierarchical timing analysis using the generated model with post-CPPR flat timing analysis on the original design.

The inputs and output files follow industrial standard formats, including gate-level netlist in Verilog (.v), net parasitics (in terms of RC network) in Standard Parasitic Exchange Format (.spef), cell libraries (including cell functionality and early/late timing information) and macro model in (restricted) Liberty (.lib).
The timing macro modeling problem is challenging because there is a tradeoff among 1) model accuracy, 2) model size (in terms of the number of internal arcs and pins), 3) model generation performance (runtime and memory), and 4) model usage performance (runtime and memory). When targeting sign-off timing, the model generation algorithm should place the highest emphasis on accuracy; subsequently, our results show a very small error bound can be achieved.

3. TIMING MACRO MODELING
In this section, we detail our timing macro model generation algorithm.

3.1 Algorithm Flow
The overall algorithm flow of our timing macro model generation is shown in Fig. 2. For achieving context independency, efficiency, and accuracy, our key idea is to retain only a small amount of necessary interface logic.

First, a design is converted into an initial timing graph (see Section 3.2). Second, interface logic is captured, whereas internal logic is removed (see Section 3.3). Third, necessary pins are marked, including pins on gates in the first few stages within the fanout cone of an input port, on gates directly connected to output ports, or on gates with multiple fanouts in the clock network (see Section 3.4). Fourth, the timing graph is iteratively reduced without touching the marked pins (see Section 3.5). Fifth, pseudo pins are inserted if a restricted Liberty model is used (see Section 3.6). Finally, for achieving high accuracy over the possible operating condition range, appropriate lookup table indices for each timing arc are determined (see Section 4), and a timing macro model is generated accordingly. Fig. 3 demonstrate key steps in our algorithm. We devise anchor pin insertion and deletion by generalizing existing graph reduction techniques. Combining with parallel merging and clock network simplification, our reduction technique effectively simplify the timing graph, and CPPR credits are well maintained.

3.2 Initial Timing Graph Construction
An initial timing graph is a directed graph constructed based on the given netlist and early/late cell libraries. For facilitating graph manipulation and timing computation, we create two nodes (one rise pin and one fall pin) for each cell pin, primary input (input port), primary output (output port), and clock source. An edge represents a timing arc between two pins, describing a timing type and a timing sense (positive, negative or non-unate [11]). Each node (rise/fall pin) is associated with its early/late timing, e.g., arrival time, required arrival time. Setup and hold times are also attached to sequential element nodes. Each edge (timing arc) is associated with early/late output slew and early/late delay lookup tables indexed by input slew and output loading.

3.3 Interface Logic Capturing
Because interface logic is sensitive to the changes on boundary conditions and our goal is to generate a compact and accurate macro model, we try to retain only a small amount of necessary interface logic. To achieve this goal, we start from interface logic capturing.

First, we perform a forward traversal starting from primary inputs except clock source pins until endpoints (primary outputs or data pins of sequential elements) are reached. The reached endpoints, the clock pins of reached sequential elements are collected. Second, we perform a backward traversal from primary outputs and from the collected points. During the backward traversal,
For example, the highlighted dots in Fig. 3(b) are preserved necessary pins of the captured interface logic in Fig. 3(a). By preserving necessary pins, we can avoid accuracy loss at subsequent graph reduction.

3.5 Timing Graph Reduction

After interface logic capturing and necessary pin preservation, we iteratively reduce the timing graph without touching the marked pins. In the following, we first briefly review existing reduction techniques and then present the generalization.

3.5.1 Existing Reduction Techniques

As shown in Fig. 4, existing reduction techniques include: Serial merging combines two consecutive timing arcs into one and eliminates the intermediate pin (Fig. 4(a)). Parallel merging combines multiple timing arcs between two pins into one (Fig. 4(b)). Tree merging replaces a tree-structured subgraph by direct arcs from leaves to the root (Fig. 4(c)). Biclique-star replacement replaces a complete bipartite graph with a star structure (Fig. 4(d)).

3.5.2 Anchor Pin Insertion and Deletion

Herein, we introduce the generalization of serial merging, tree merging, and biclique-star replacement. Fig. 4(e) illustrates the concept of anchor pin insertion and deletion. First, we discuss anchor pin deletion. The deletion gain of pin $p_i$ can be computed as

$$\text{delgain}(p_i) = \text{indegree}(p_i) + \text{outdegree}(p_i) - \text{indegree}(p_i) \cdot \text{outdegree}(p_i),$$

(1)

where indegree means the number of incoming timing arcs of $p_i$, whereas outdegree means the number of outgoing timing arcs of $p_i$. When a deletion gain is negative, anchor pin deletion increases the number of timing arcs, this is undesired. Interestingly, in most cases, we observe that the accuracy is slightly improved for zero deletion gain in our experiments. Hence, we perform anchor pin deletion when a pin has a nonnegative deletion gain (delgain $\geq 0$). Anchor pin insertion is the reverse form of anchor pin deletion; the insertion gain can be defined similarly.

It can be seen that serial merging is a special case of anchor pin deletion; tree merging can be obtained by iteratively performing anchor pin deletion on a tree-structured subgraph; biclique-star replacement is a special case of anchor pin insertion.

3.5.3 Clock Network Simplification

We simplify the clock network by necessary pin preservation plus iterative anchor pin deletion. According to the way we preserve necessary pins in the clock network, the CPPR effect can be captured well.

3.6 Optional: Pseudo Pin Insertion & Sharing

There is an assumption for the used Liberty format in TAU 2016 timing contest [10]: Only one timing arc is allowed between two pins, i.e., one-to-one correspondence between a timing arc and its corresponding pins. The reduced timing graph obtained in Section 3.5 may not be compatible with this assumption. Therefore, we present pseudo pin insertion and sharing.

In the reduced graph, the connections between two pins can be classified into fifteen types (Fig. 5), where each pin corresponds to one rise pin and one fall pin. With a special encoding, types 1, 2, 4, 8 are primitive types. Any of the remainder is composed of two or more primitive types, e.g., type 12 is composed of types 4 and 8.

Seven types listed in Fig. 5(b) are incompatible with the restricted Liberty format. Therefore, we create a pseudo pin to break a timing arc in each incompatible type. Because an inserted pseudo pin corresponds to one rise pin and one fall pin, either only the
rise pin (Fig. 6(a)) or only the fall pin (Fig. 6(b)) of an added pin is used. We further share pseudo pins between these two groups to reduce the total number of inserted pseudo pins, i.e., types 10 and 5 are shared, types 14 and 7 are shared, types 13 and 11 are shared, and type 15 is shared itself.

4. LOOKUP TABLE INDEX SELECTION

In this section, we determine our lookup table indices to minimize the interpolation error when the generated model is used.

As mentioned in Section 3.4, the shielding effect of slew propagation makes slew values converge into a user defined tolerance range after few stages; the output loading for gates unconnected to primary outputs is constant (obtained from the given net parasitics). The lookup table of a mergeable timing arc (converged slew and constant loading) can be easily computed.

In the following, we focus on the part of interface logic affected by variant operating conditions—timing arcs connecting to preserved pins (unconverged slew or variant loadings).

4.1 Wire Timing Arc Indexing

A wire timing arc is associated with delay and output slew. The delay is directly computed based on its parasitic RC tree, and the delay value is independent of slews. Herein, we discuss selecting delay for output slew lookup tables on a slew-unconverted wire timing arc.

According to [12], the output slew can be approximated by a function \( f(x) = \sqrt{x^2 + c^2} \)

where \( x \) means input slew, \( c \) is contributed by the second moment of the slew. A more sophisticated approximation can also be used. Our goal is to select \( n \) most significant points from the input slew range \([x_0, x_{n+1}]\) such that the difference between the area under curve \( f \) and the area under piece-wise linear function via these selected points is minimized. The line segment connecting \( x_i \) and \( x_{i+1} \) can be written as:

\[
L_i(x) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} (x - x_i) + f(x_i), x \in [x_i, x_{i+1}].
\]

The area difference between approximated lines and \( f \) is:

\[
\sum_{i=0}^{n} \int_{x_i}^{x_{i+1}} (L_i - f(x)) dx.
\]

By taking gradient on Equation (4),

\[
\nabla \sum_{i=0}^{n} \int_{x_i}^{x_{i+1}} (L_i - f(x)) dx = 0.
\]

Consider initially sampled points \( x_0 < x_1 < x_2 < \ldots < x_n < x_{n+1} \). The minimum area difference occurs when \( x_i \) moves to \( x'_i \),

\[
x'_i = c \sqrt{\frac{m^2}{1 - m^2}},
\]

where \( m \) is the slope of a straight line connecting \((x_{i-1}, f(x_{i-1}))\) and \((x_{i+1}, f(x_{i+1}))\), \( m < 1 \).

We can start with a set of uniformly sampled indices and then obtain final indices by iteratively applying the calculation of Equation (6) until each index value stabilizes within a small enough region. If the stable value is outside \([x_0, x_{n+1}]\), \( x_0 \) and \( x_{n+1} \) are used instead.

4.2 Cell Timing Indexing

Cell timing contains delay, output slew, and timing constraints. Delay and output slew depend on only the input slew of a related pin because output loading is fixed (directly obtained from output net parasitics). Given the minimum and maximum propagated slews of a related pin, cell timing indices can be defined as the minimum and maximum propagated slews and the original indices covering the interval of minimum and maximum propagated slews. For example, if a cell has original slew indices as \{1, 3, 5, 10, 20, 30, 100\}, the respective minimum and maximum propagated slews of this cell are 6 and 15, then the selected indices for this cell timing arc are \{3, 5, 10, 15, 20\}. On the other hand, timing constraints depend on the slews on both constrained pin and related pin. Thus, we adjust both indices similarly.

4.3 Output Cell Arc Indexing

A wire connected to an output port has a linear response to output loading variation. The exact loading at an output port is unknown during macro modeling. We merge the cell timing arc and wire connection together in our model (Fig. 7), and query the output cell timing according to the output loading context.

\[
cell_{ex}(C_L) = cell_{ori}(C_L + C_N) + wire_{ori}(C_L + C_N), (7)
\]

where \( cell_{ex} \) is cell timing after merged, \( cell_{ori} \) is the original cell timing, \( wire_{ori} \) means wire RC timing, \( C_L \) is the output loading, and \( C_N \) is the net lumped capacitance. The net lumped capacitance is viewed as an internal capacitance which is invisible in the generated model; thus, we shift down the original cell indices by an offset (equal to \( C_N \)) as our indices for output loading. Notably, output cell timing still depends on input slew and output loading, and thus cell timing indexing described in Section 4.2 is applied here to reduce slew indices.

5. EXPERIMENTAL RESULTS

We implemented our algorithm in the C++ programming language and compiled it with g++ 4.8.2. We executed the program on a platform with two Intel Xeon 3.5 GHz CPUs and with 64 GB memory.

Experiments were conducted on the TAU 2016 timing contest on macro modeling benchmark suite released by [10] as listed in Table I. ‘#PIs’ denotes the number of primary inputs, ‘#POs’ the
number of primary outputs, ‘#Gates’ the number of gates, ‘#Nets’ the number of nets, and ‘Total’ the sum of #Gates and #Nets.

Fig. 8 shows the experimental framework. The operating conditions for model generation are that any input slew ranges from 5 ps to 250 ps, any output loading ranges from 5 fF to 250 fF, and input delay ranges from 0 ps to 2000 ps (for evaluating the correctness of the CPPR effect handling). Two macro models are generated, one for early timing and one for late timing. The baseline is post-CPPR flat timing analysis on the original design. Overall, our algorithm outperforms LibAbs in model accuracy, model size, model generation performance (runtime and memory), and model usage performance (runtime and memory).

On average, compared with LibAbs, our algorithm can improve model accuracy, model file size, model generation runtime, model generation memory, model usage runtime, and model usage memory by 84%, 74%, 47%, 71% 67%, 63% reduction, respectively (see ‘Average Ratio 1’). First, our model is extremely accurate: The maximum error is much less than 0.1 ps, implying that our algorithm maintains context independency and handles the CPPR effect very well. Second, considering our algorithm is single-threaded, whereas LibAbs is multi-threaded, our model generation performance (runtime and memory) is superior (only 53% and 29% of LibAbs). Usually, there are many duplicate blocks on a chip, and hierarchical timing analysis is rerun many times; thus, the model generation effort can be amortized, and the overall performance is promising—the incentive of hierarchical timing analysis.

It can be seen that our macro model is compact and accurate, thus facilitating efficient and accurate hierarchical timing analysis. In design. Overall, our algorithm outperforms LibAbs in model accuracy, model size, model generation performance (runtime and memory), and model usage performance (runtime and memory).

We compared our algorithm with TAU 2016 contest winner, LibAbs, to demonstrate its effectiveness and efficiency. The program of LibAbs was provided by the winning team and executed on the same platform as described above. LibAbs and our algorithm are both evaluated by 1) model accuracy, 2) model file size, 3) model generation performance (runtime and memory), and 4) model usage performance (runtime and memory). Table II lists the detailed comparison. ‘Max Error’ reflects model accuracy, which is measured by the maximum difference between the baseline timing report and the timing report conducted by hierarchical timing analysis using the generated macro model. ‘Model File Size’ reflects model size, which is measured by the file size of a generated macro model. ‘Generation Runtime’ represents model generation time, while ‘Generation Memory’ means model generation memory requirement. ‘Usage Runtime’ and ‘Usage Memory’ are similarly defined for hierarchical timing analysis. ‘Ratio’ means the ratio of our result to LibAbs for each benchmark design. Overall, our algorithm outperforms LibAbs in model accuracy, model size, model generation performance (runtime and memory), and model usage performance (runtime and memory). On average, compared with LibAbs, our algorithm can improve model accuracy, model file size, model generation runtime, model generation memory, model usage runtime, and model usage memory by 84%, 74%, 47%, 71% 67%, 63% reduction, respectively (see ‘Average Ratio 1’). First, our model is extremely accurate: The maximum error is much less than 0.1 ps, implying that our algorithm maintains context independency and handles the CPPR effect very well. Second, considering our algorithm is single-threaded, whereas LibAbs is multi-threaded, our model generation performance (runtime and memory) is superior (only 53% and 29% of LibAbs). Usually, there are many duplicate blocks on a chip, and hierarchical timing analysis is rerun many times; thus, the model generation effort can be amortized, and the overall performance is promising—the incentive of hierarchical timing analysis.

It can be seen that our macro model is compact and accurate, thus facilitating efficient and accurate hierarchical timing analysis. In
fact, when accuracy is a major concern, there is no imperative need to generate macro models for relatively small blocks (e.g., mgc and vga_lcd in our experiments). Compared with the baseline performance, our model usage performance leads to average 27% and 43% reduction on runtime and memory, respectively (see ‘Average Ratio 2’). Generally, hierarchical timing analysis using our model can significantly reduce runtime and memory compared with flat timing analysis on the original design.

Table III shows the effectiveness of our graph reduction technique on model size. ‘Ours: Interface Logic’ means our algorithm without performing graph reduction (i.e., keeping the whole interface logic), while ‘Ours: Final’ means the complete version of our algorithm. Our graph reduction technique contributes over 77% reduction on model file size; we retain only a very small amount of interface logic in our model. It can be seen that our macro model is very compact and accurate.

6. CONCLUSION
To achieve efficient and accurate hierarchical timing analysis, in this paper, we propose compact and accurate timing macro modeling. Our key idea is to make our macro model contain only a small amount of interface logic and maintain high accuracy. To generate a compact model, we generalize existing graph reduction techniques to anchor pin insertion and deletion. To generate an accurate model, we preserve necessary pins and wisely select proper index values of lookup tables to describe timing arcs. Our experiments are conducted on the TAU 2016 timing contest on macro modeling benchmark suite. Experimental results show that our algorithm delivers superior efficiency and accuracy: Hierarchical timing analysis using our model can significantly reduce runtime and memory compared with post-CPR flat timing analysis on the original design. Moreover, our algorithm outperforms TAU 2016 contest winner in model size, model accuracy, and model usage performance (in terms of runtime and memory). Future work includes expediting our model generation by distributed computing, developing a new format to facilitate model generation and usage, and considering coupling effects in hierarchical timing analysis.

7. REFERENCES
[6] Liberty: The semiconductor industry’s most widely used library modeling standard. Available at: https://www.opensourceliberty.org/