AN EFFICIENT AND ACCURATE TIMING MACRO-MODELING ALGORITHM FOR LARGE HIERARCHICAL DESIGNS

BY

TIN-YIN LAI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Martin D. F. Wong

ABSTRACT

Ever-increasing circuit design complexity is driving the need for fast and accurate macro-modeling algorithms to accelerate hierarchical timing. We introduce *LibAbs*, an effective macro-modeling algorithm that efficiently supports high accuracy, high compression rate, and multi-threading. LibAbs applies tree-based graph reduction techniques to reduce the model size with accuracy values comparable to those of the flat model under a multi-threaded environment. LibAbs outperforms existing tools including the top winners from the TAU 2016 macro-modeling contest in terms of model size, accuracy, and runtime on industry benchmarks. The in-context usage of our abstracted model has also demonstrated promising performance for timing-driven optimizations in large hierarchical designs.

ACKNOWLEDGMENTS

This work is sponsored by the National Science Foundation under Grant CFF-1320585 and CFF-1421563. I appreciate my advisor, Prof. Martin D F Wong, and all the people in the CAD group of UIUC ECE for the discussion. I acknowledge the TAU 2016 Timing Contest committees for the discussion, and team iTimerM for their binary.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

As design complexities continue to increase, timing analysis has been one of the most time-consuming tasks in the optimization cycles recently. Hierarchical timing analysis is one of the solutions to speed up the timing closure via precomputing timing in several parts of designs. In hierarchical timing, a large design is partitioned into several manageable sub-designs. These manageable sub-designs can generate timing in shorter runtimes. By optimizing timing on these sub-designs, the timer can reduce the computational space in timing analysis on large designs. Timing macro-modeling is an essential step in the hierarchical timing to optimize runtime. A timing macro-modeler, the engine of timing macro-modeling, abstracts sufficient timing behavior of sub-designs into macro-models. A timing macro-modeler is shown in Figure 1.1. The timing of sub-designs can be efficiently reproduced using macro models in the timing analysis of large designs. A successful macro-model is small, accurate, and reusable. Hierarchical timing analysis with successful macro-models significantly accelerates the optimization cycles by saving runtime on identical and duplicated sub-designs.

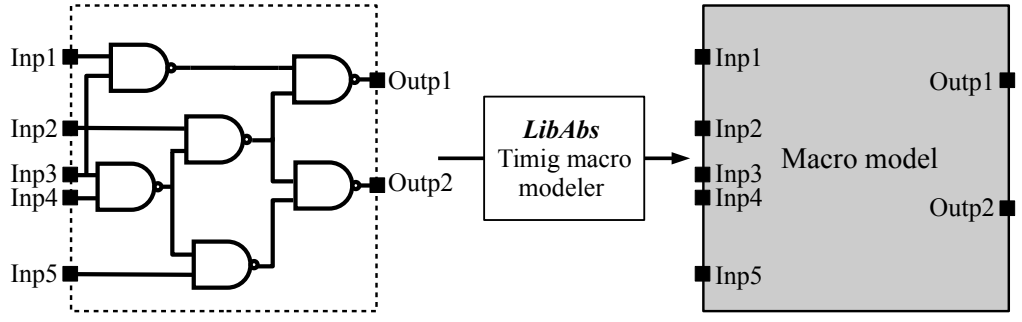


Figure 1.1: A timing macro-modeler abstracts timing behavior of a sub-design into a macro-model to speed up the timing analysis.

However, in prior works, macro-models were unable to capture accurate timing [1]. In [1], [2], and [3], the authors generated macro-models based

on a comparably straightforward timing model without considering parasitic delay. To date, timing macro-modeling in large hierarchical timing is still an open problem. Academy and industry jointly held the 2016 TAU timing contest [4] to seek novel solutions. According to the published results from the TAU timing contest [4], the top performers were unable to strike a balance between model size and accuracy. Therefore, we provide a new algorithm to address the problem in both accuracy and model size.

In this thesis, we introduce LibAbs, a timing macro-modeling algorithm, that efficiently supports:

- **Industry standard format.** LibAbs is compatible with the industry standard format. Our macro-model can be integrated into the existing timing engine.
- **Accuracy.** Compared with the timing in the original flat circuit, LibAbs generates accurate macro-models.
- **High compression rate.** LibAbs generates macro models with small model size.
- **Multi-threading.** LibAbs supports multi-threading to generate a macro-model in parallel efficiently.
- **Effective macro-usage.** To facilitate timing-driven optimizations, our macro-models reduce the total runtime of both the in-context and out-of-context timing analysis.

The above advantages make LibAbs an accurate, efficient, and high-quality macro-modeler. Compared to the original flat timing, the experimental results demonstrate that the macro-model generated by LibAbs is smaller by 33% while the performance margin in terms of accuracy is kept within 0.3 ps. In addition, in-context usage can speed up to around $\times 3$ with hundreds of operations.

The rest of the thesis is organized as follows. In Chapter 2, we formulate the problem. In Chapter 3, we show the theories and strategies of LibAbs. Our experimental results are in Chapter 4. Finally, in Chapter 5, we conclude our accomplishments.

CHAPTER 2

PROBLEM FORMULATION

We follow the rules of TAU 2016 timing contest [4].

Problem 1 *The goal of timing macro-modeling is to create a macro-model from a given circuit design and the boundary timing. The boundary timing includes primary input arrival time, primary input slew, primary output required arrival time, and primary output load. In the TAU contest [4], the range of primary input slew is 0 ps to 250 ps and primary output load is 0 pf to 250 pf. A macro-model is capable of reproducing the timing behavior at primary inputs and primary outputs. A macro-model is written in the format of a single cell in Liberty files.*

As shown in Figure 2.1, the timing macro-modeler generates a library cell that represents a full netlist described in input files. To validate accuracy, we use both full netlist and macro-model in the same parent-level circuit and compare the timing on input/output ports using OpenTimer [5], the golden timer in the TAU contest [4]. In addition, our macro-models also consider the parasitic delay and Common Path Pessimism Removal (CPPR).

Hereafter, we first introduce input files in Section 2.1. Section 2.2 describes the output files and evaluation part.

2.1 Input Files

Our input files follow the industry formats. A gate-level netlist is defined in a Verilog ($*.v$) file. Parasitic information, provided in the Standard Parasitic Exchange Format ($*.spef$) file, describes resistor-capacitor (RC) trees on wires for estimating the Elmore delay. An assertion ($*.timing$) file sets initial boundary timing, including arrival time and slew at primary inputs, output load and required arrival time at primary outputs. Liberty ($*\_Early.lib$ and

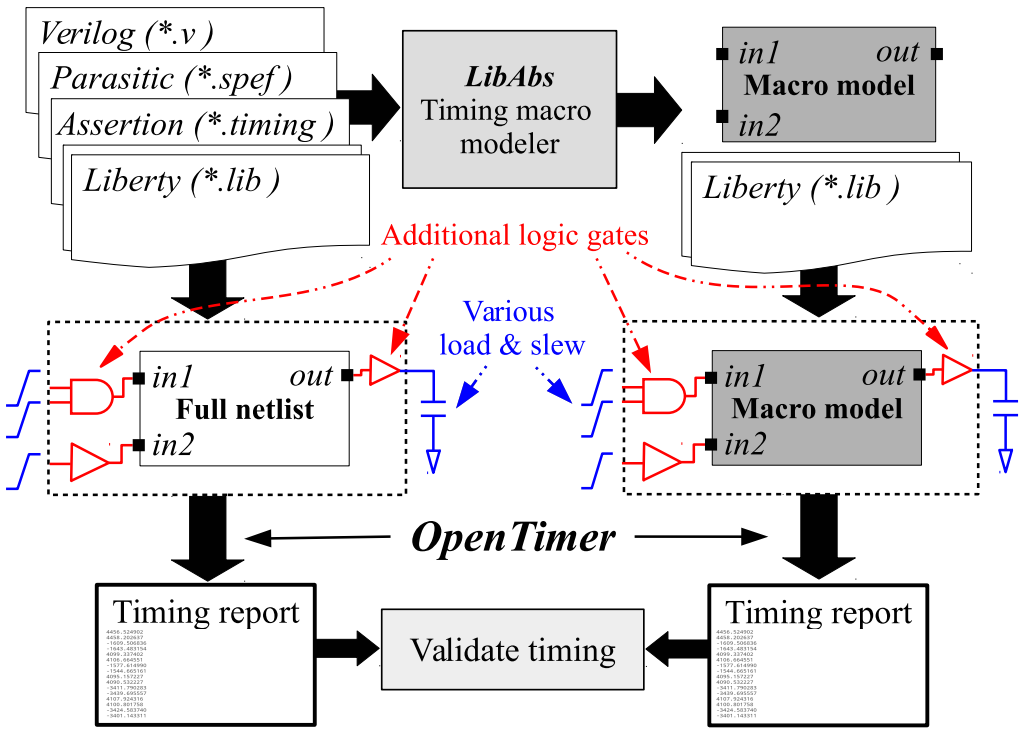


Figure 2.1: The flow of timing macro-modeling.

$*\_Late.lib$) files provide the Early and the Late timing information of standard cells for hold tests and setup tests respectively. Timing information in Liberty files includes delay, slew, and setup/hold constraints in the form of look-up tables (LUTs).

2.2 Output Files and Evaluation

Output files are two liberty (Early and Late) files that contain a single cell representing the macro-model. For all the primary inputs/outputs of the original netlist, there exist corresponding primary input/output pins in this single cell. Liberty files save timing information in 2-D LUTs.

For evaluating the accuracy, the original netlist is connected with some additional gates to input ports and from output ports. The assertion file provides input slew/arrival time and output load/required time for OpenTimer to report slack at both input and output pins, and arrival time at output ports. The output of OpenTimer is the golden timing report. The macro-model connects to the same additional gates at both input pins and output pins as well. We set the same assertion file and report timing. Finally,

we compare the output timing report with the golden timing report. We also compare the runtime and memory usage in the two timing analyses.

CHAPTER 3

ALGORITHM

In this chapter, we introduce the algorithm of LibAbs. Overall program flow is shown in Figure 3.1.

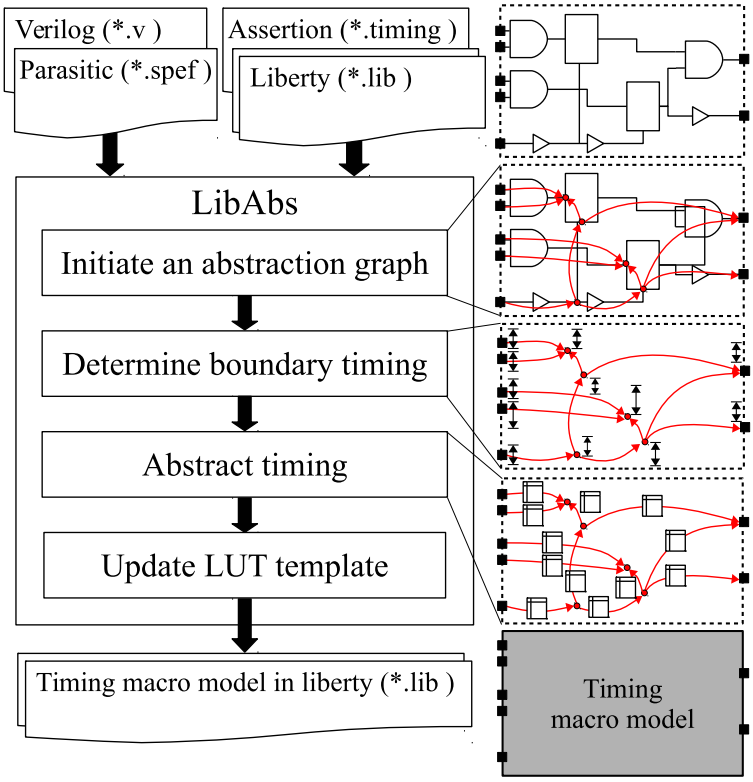


Figure 3.1: LibAbs program flow.

In LibAbs, we first initiate an abstraction graph, G_{abs}, from the original timing graph G_{timing}. We use the terms $abs-node$ and $abs-edge$ to refer to the node in G_{abs} and the edge in G_{abs} respectively. Secondly, to reduce the search space and computational effort, we determine boundary timing on all the abs-nodes. Third, considering the boundary timing, we abstract timing information path by path. Finally, we update LUT template for library syntax usage and write out library files.

3.1 Abstraction Graph Construction

Based on an original timing graph, we construct an abstraction graph by merging timing arcs. We aim to merge timing arcs with minimum accuracy loss. The merging of branch-in/out arcs may induce accuracy loss. However, if a sub-graph of a timing graph is a tree structure, there exists a sub-abs-graph that consists of abs-nodes on a root and leaves of the tree graph, and abs-edges link the root to each leaf respectively. Furthermore, if the load capacitance value remains fixed on each node, the abs-graph merges timing with no accuracy loss.

The values of load capacitance on internal pins that are not connected to the primary outputs are constants because the sub-design circuit remains unchanged in the parent-level optimization cycles. If we can find a sub-graph of a timing graph which is a tree, we can accumulate delay and derive slew with a given slew and fixed output loads trough a unique path between a root and each leaf on each abs-edge. This is suitable in both out-trees, where the leaf pins are in the fanout cone of a root pin, and in-trees, where the leaf pins are in the fanin cone of a root pin. However, in the netlist, the number of branch-ins is usually more than the number of branch-outs because most of the standard cells are equipped with branch-in timing arcs, including NAND, AND, NOR, OR, XOR, etc. Therefore, to minimize the model size, we would like to find all the in-trees with the deepest possible leaves, or maximal in-trees.

We are unable to annotate load capacitance on internal pins in liberty files. However, load capacitance values vary at timing arcs that are connected to the output ports, as the macro-block can be reused in different places of the design. For wires that are only connected to a single output port, load indices on LUTs can solve the problem. Some wires connect not only to output ports but also to cell pins. In this case, load capacitance values on output ports vary, but internal cell pins are unaware of it. To solve this problem, we connect all input pins of previous cells to the primary outputs and other internal cell pins. We call these abs-edges *primary output segments*.

A timing graph is a directed acyclic graph (DAG). The problem is to partition a DAG into a forest of maximal in-trees. By definition, in an in-tree, all nodes have less than one output edge, not counting the root. Therefore, the node with multiple branch-out abs-edges must be the root of the in-tree.

To find the longest paths in the in-trees, we cut DAG at multiple branch-out nodes. We found out that depth-first search (DFS) can complete this task. We will show how LibAbs abstracts the timing graph into an abstraction graph G_{abs} in Algorithm 1.

Algorithm 1 Abstraction graph construction

Input: a timing graph G_{timing}
Output: an abstraction graph G_{abs}

```
initialization
insert all primary inputs/outputs to G_abs
push all primary inputs in a queue
mark all primary inputs as visited
while queue not empty do
    u ← pop the top of queue
    for all v ← descendant of u do
        while number of v's descendant ≤ 1 do
            v ← descendant of v
        end while
        if v connects to primary output & cell pin then
            insert an abs-edge from u to v into G_abs
            for all n ← branch-out nodes from v do
                insert an abs-edge from v to n into G_abs
                if n is not visited then
                    push n in queue
                    mark n as visited
                end if
            end for
        else
            v ← descendant of v
            if v is not visited then
                push v in queue
                mark v as visited
            end if
            insert an abs-edge from u to v into G_abs
        end if
    end for
end while
```

Figure 3.2 shows an example circuit. Originally, there are 20 edges in the timing graph. We construct G_{abs} as follows. In this circuit, the blue part corresponds to an in-tree. We connect leaves, i.e., the primary input A, the primary input B, and Q pin of DFF, to the root pin, i.e., D pin of the DFF.

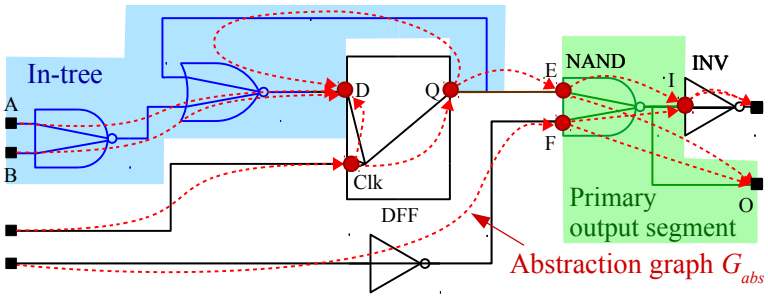


Figure 3.2: An abstraction graph example. We construct G_{abs} with 13 abs-edges and 12 abs-nodes including primary inputs and primary outputs.

Next, the green part of the circuit shows a primary output segment, where we connect the E pins and the F pin of the NAND gate to the primary output O and the I pin of the INV gate. In the end, there are only 13 abs-edges in G_{abs}. The final G_{abs} of the circuit is shown as red dashed lines. In addition to the input and output ports, 6 internal abs-nodes are shown as red dots.

3.2 Boundary Timing Determination

Because of the design constraints, a macro-model is usually used under certain boundary timing defined by a range of input slew and output load. Based on the boundary timing, the macro-modeler can reduce the search space by limiting the size of LUTs in the following steps. According to the TAU timing contest [4], the range of input slew is from 0 ps to 250 ps and the range of output load is from 0 pf to 250 pf. Therefore, we record both the minimum and the maximum of slew and load on abs-nodes by setting the best and the worst slew and load on all input/output ports. Properly constraining the boundary timing is beneficial for reducing the search space.

3.3 Timing Abstraction

In timing abstraction, LibAbs tabulates timing information. In the first step of timing abstraction, we assign a set of proper indices to each abs-edge. Secondly, we trace through timing arcs on abs-edges to derive timing for every index entry. Finally, we assign the timing into the corresponding LUTs. The timing here includes delay values, slew values, or constraint values.

Algorithm 2 Determine boundary timing

Input: a timing graph G_{timing}
Input: an abstraction graph G_{abs}
Output: an abstraction graph G_{abs}

```
1: set min input slew on G_timing
2: set min output load on G_timing
3: update timing on G_timing
4: record min slew and min load on all abs-nodes
5: set max input slew on G_timing
6: set max output load on G_timing
7: update timing on G_timing
8: record max slew and max load on all abs-nodes
```

Algorithm 3 Abstract timing

Data: an abstraction graph G_{abs}

```
1: for each abs-edge on G_abs do
2:     initiate indices
3:     for each index entry on LUTs do
4:         infer timing
5:         assign timing to LUTs
6:     end for
7: end for
```

3.3.1 Indices Initiation

Assigning indices is critical in timing macro-modeling. If the indices' values are not properly selected, distortion in accuracy leads to an inaccurate timing model. For transition tables and delay tables, two indices are slew on the source pin and load on the sink pin. In constraint tables, two indices are slew on the clock pin and slew on the data pin of a flip-flop.

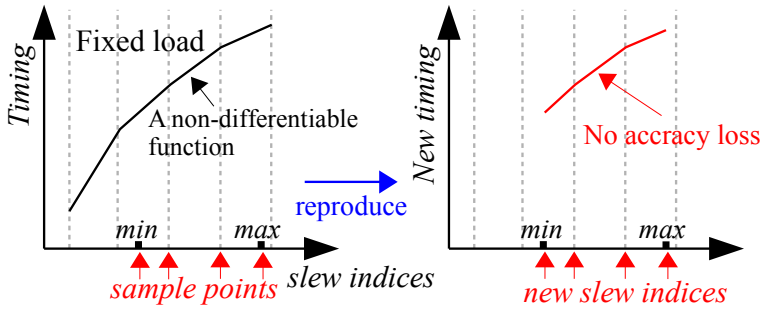


Figure 3.3: Sampling a non-differentiable function.

First, we select sets of indices on transition tables and delay tables. In the

timing analysis, timings on wires are continuous functions because parasitic delay and slew are based on the Elmore delay model. Tabulating parasitic timing is equivalent to sampling a smooth and differentiable function. However, the timer interpolates a LUT with a set of slew and load on a cell arc. LUT is a non-differentiable function. Actually, except for the non-differentiable points, LUT is a linear function [4]. Therefore, if we sample on non-differentiable points, we lose no accuracy. The idea is illustrated in Figure 3.3. To avoid accuracy distortion on timing, LibAbs selects slew indices that are derived from the first cell arc from the source node because the slew indices from the first cell arc determine the first set of non-differentiable points. If the first arc from the source node is a wire, we have to find the first cell arc and back-propagate indices to the source node. The formula for back-propagating slew is derived from the Elmore delay if load capacitance values remain constants:

$$slew\_indices_{source} = \sqrt{slew\_indices_{sink}^2 - impulse^2} \tag{3.1}$$

where the impulse is the impulse delay from the Elmore model [4]. In addition, by utilizing the boundary timing from Section 3.2, we bound indices to reduce the size of LUTs.

Similarly, load indices are determined based on the last cell arc to the sink node. If the last arc is not a cell arc, we need to forward propagate load indices as well.

$$load\_indices_{sink} = load\_indices_{last\_arc} - total\_cap_{wire} \tag{3.2}$$

We bound the load indices as well. For an internal pin, the load is fixed.

For instance, in Figure 3.4, an abs-edge starts from a source node to a sink node. We derive slew indices of this abs-edge from cell arc b (1.0, 2.0, 3.0). We back propagate slew indices from Equation 3.1 and get slew indices (0.86, 1.94, 2.96). Finally, we insert the maximum and minimum slew into slew indices and remove indices outside the boundary. We remove 0.86 and 2.96 and insert 1.2 and 2.5 into slew indices. Slew indices of the abs-edge are (1.2, 1.94, 2.5). Load indices are derived from cell arc d and back-propagate load. However, load indices are finally bounded by 8.2 in this case.

Two indices of a constraint table are slew from the clock pin and the data

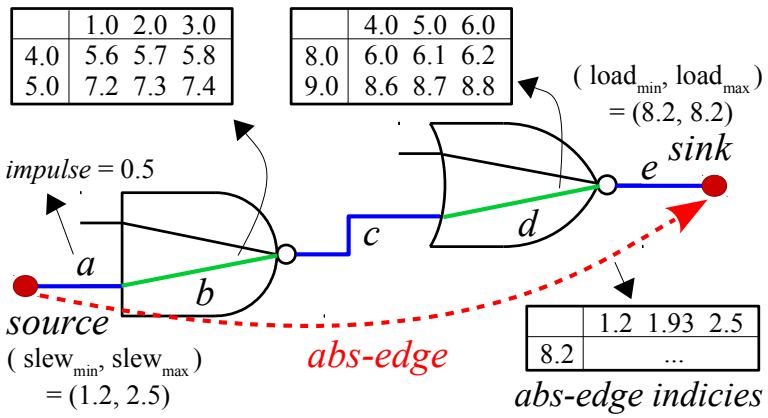


Figure 3.4: Initiate indices. Slew indices of the abs-edge (1.2, 1.94, 2.5), are derived from b. Load indices (8.2) are derived from cell arc d.

pin of a flip-flop. A clock pin has two branch-out abs-edges. A data pin of a flip-flop is the end of the timing path. Therefore, in G_{abs}, the abs-edge with constraint table copies indices from constant arcs and bound slew indices by slew boundary on the clock pin and the data pin.

3.3.2 Timing Inference

After indices have been decided, LibAbs infers timing for every index entry of the LUTs on each abs-edge by tracing through cell arcs and RC arcs on abs-edges. However, tracing through arcs on a given abs-edge with a sink node and a source node is difficult because of the multiple branch-out arcs from a source node and multiple branch-in arcs to the sink node. Therefore, for implementation, we annotate a directed node on an abs-edge to direct which branch-out arcs to trace. The directed node is at the sink node of the first arc.

As we aim to create a single cell to model a circuit, there are only two types of internal cell arcs: (1) combinational arcs and (2) constraint arcs. We first discuss delay and slew on combinational arcs. To propagate delay with a pair of input slew and output load, we accumulate delay and update current slew as we trace through timing arcs from the source node to the sink node of an abs-edge. As we trace through cell arcs, we interpolate the LUTs from the original standard cell library to get delay and slew. We derive delay and slew based on the Elmore delay and the parasitic wire information given in .spef files. In constraint arcs, we derive slew from a pair of given clock pin

slew and data pin slew. We show the details in Algorithm 4.

Algorithm 4 Infer timing

Data: an *abs-edge*
Input: slew $slew_{source}$ on source and load $load_{sink}$ on sink
Output: *delay* and *slew*

```
1: u ← the source node of the abs-edge
2: v ← the directed node of the abs-edge
3: e ← the timing arc from u to v
4: delay ← 0
5: slew ← slew_source
6: while v! = sink of the abs-edge do
7:     if e is a cell arc then
8:         delay ← d + interpolated delay on e
9:         slew ← interpolated slew on v
10:    else if e is a RC arc then
11:        delay ← d + delay derived from e
12:        slew ← slew derived to v
13:    end if
14:    e ← the next timing arc
15:    v ← the sink node of e
16: end while
```

3.4 Update LUT Template

In timing analysis, the LUT template, including LUT name, size, and the range of variables in indices, is essential information for the timer. As we already tabulated timing in the previous steps, we need to insert LUT templates in liberty files. Therefore, we sweep through all the LUTs on abs-edges and insert the corresponding LUT templates.

3.5 Multi-threading

It is desired to utilize the power of many-core machines to develop a parallel algorithm. LibAbs strongly supports multi-threading in timing abstraction which involves heavy computations. In Section 3.3, the variables for inferring timing on abs-edges are independent of each other. Therefore, multiple threads can spawn to deal with each iteration of inferring timing on

an abs-edge in a parallel manner to improve the throughput. The higher throughput also translates into higher speedup. With the support of multi-threading, LibAbs generates macro-models efficiently. In our program, the multi-threading version with 8 threads computing is more than $\times 4$ faster than the single-threading version. As shown in Algorithm 5, we apply multi-threading in Algorithm 3.3.

Algorithm 5 Abstract timing with multi-threading

Data: an abstraction graph G_{abs}

```
#pragma omp parallel for
for each abs-edge on G_abs do
    initiate indices
    for each index entry on LUTs do
        infer timing
        assign timing to LUTs
    end for
end for
```

CHAPTER 4

EXPERIMENTAL RESULTS

We implement LibAbs in C++ language with OpenMP 3.1 as multi-threading library [6]. In the experiment, our machine is equipped with Intel(R) Xeon(R) CPU E5-2660 @ 2.20GHz with 8 cores and 128GB memory on Linux 64-bits system [7]. Our benchmarks are from the 2016 TAU timing contest [4]. We compare with the top-two performers from the 2016 TAU timing contest [4].

4.1 Abstraction Runtime and Memory Usage

In the beginning, we compare the abstraction runtime and memory usage with the top performers from the TAU timing contest 2016 [4]. We collect runtime and memory data in the average of 20 runs. Table 4.1 shows that runtime of the 2^{nd} place team runs $\times 3.9$ to $\times 9.5$ slower than our work. In addition, memory usage of the 2^{nd} place team is $\times 1.4$ to $\times 2.3$ higher than our work. The macro-modeler from the 2^{nd} place team is not multi-threaded, and their throughput is low. The experimental results of runtime and memory usage show that our work is faster and more efficient in memory usage compared to the top performers in the TAU timing contest. It is expected that the new liberty files need longer time for setting up in OpenTimer [5] because the liberty file size is larger.

Table 4.1: Runtime and memory usage

Circuits	# of Gates	Our work		1^{st} of TAU contest		2^{nd} of TAU contest	
		runtime	memory	runtime	memory	runtime	memory
mgc_edit_dist	221539	13.51 s	1.68 GB	28 s	1.80 GB	67 s	4.02 GB
vga_lcd	286413	18.65 s	2.27 GB	32 s	2.19 GB	75 s	4.55 GB
leon3mp	1534156	109.64 s	11.85 GB	317.5 s	14.67 GB	419 s	17.10 GB
netcard	1628325	117.31 s	12.49 GB	341 s	15.37 GB	1117 s	23.00 GB
leon2	1892057	136.66 s	14.97 GB	409.5 s	18.26 GB	926.5 s	32.35 GB

4.2 Accuracy

For testing accuracy, we experiment on the benchmarks with *.timing1 and *.timing2 file to set up boundary timing from the final evaluation of TAU 2016 and report **(1) slack at primary inputs**, **(2) slack at primary outputs**, and **(3) arrival time at primary outputs**. We estimate the accuracy in error values by averaging over both slack and arrival time from *.timing1 and *.timing2. A time unit in the original circuit library is 1 ps. In Table 4.2, our max error in all benchmarks is less than 0.26 ps with input slew and output load in range of 0 ps to 250 ps and 0 pf to 250 pf respectively. Moreover, our average error in all benchmarks is less than 0.06 ps. Comparing to the 1st place team, our model results in similar error values within 0.04 ps. In the benchmarks vga_lcd and mgc_edit_dist, maximum error values of the 2nd place team are similar to our work. However, in benchmark leon3mp, netcard, and leon2, the max error values result in ×154.13 to ×1132.07 more error than our work. The macro-model from the 2nd place team is not accurate. Our macro-models, on the other hand, are accurate and reliable for use in large hierarchical designs.

Table 4.2: Accuracy

Circuits	Our work		1st of TAU contest		Compare	
	avg.	*max.*	*avg.*	*max.*	*avg.*	*max.*
mgc_edit_dist	0.052102	0.244141	0.043471	0.215088	83.43 %	88.10 %
vga_lcd	0.005850	0.177490	0.005077	0.176269	86.80 %	99.31 %
leon3mp	0.018544	0.259522	0.015853	0.293701	85.49 %	113.17 %
netcard	0.019864	0.167968	0.026264	0.156250	132.22 %	93.02 %
leon2	0.019837	0.167969	0.017137	0.140624	86.39 %	83.72 %

Circuits	2nd of TAU contest		Compare	
	avg.	*max.*	*avg.*	*max.*
mgc_edit_dist	0.058285	0.248535	111.87 %	101.80 %
vga_lcd	0.008356	0.176758	142.84 %	99.59 %
leon3mp	0.391135	39.999390	2109.23 %	15412.72 %
netcard	0.069597	85.422486	350.36 %	50856.40 %
leon2	0.893399	190.153076	4503.65 %	113207.24 %

4.3 Compression Rate

Compared to the original circuit, our tool compresses model size into less than 33% in the number of edges and 19% in the number of nodes. Our work

largely reduces model size. In Table 4.3, the results of the compression rate also show that our macro-model is about ×3 smaller than the macro-model from the 1st place team in the number of nodes. In our experiment, our new model needs about ×1.5 to ×2 more timer setup time than the original circuit. However, due to the smaller model size, we can reduce the runtime of timing propagation by 70%. Our macro-models are highly compressed to speed up the timing analysis in parent-level designs.

Table 4.3: Compression rate

Circuits	Original circuit		Our work		Compression rate	
	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$
mgc_edit_dist	581319	691863	95288	211461	16.39%	30.56%
vga_lcd	768050	894826	129240	273016	16.83%	30.51%
leon3mp	4167632	4830700	753406	1525362	18.08%	31.58%
netcard	4458141	5264603	783831	1688054	17.58%	32.06%
leon2	5179094	5974414	949427	1894248	18.33%	31.71%

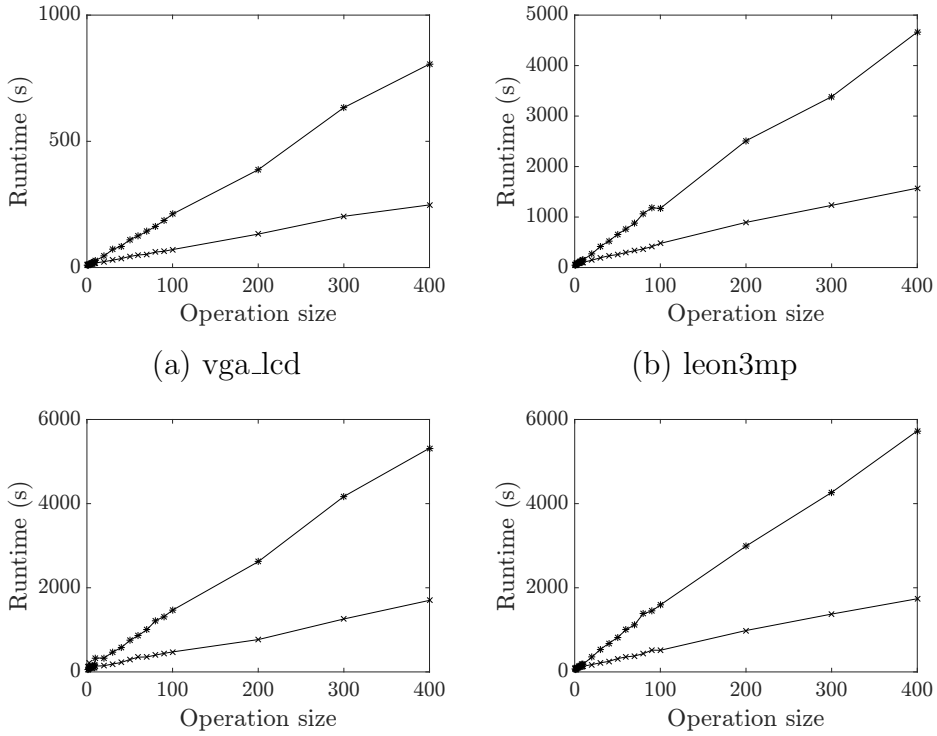
Circuits	1st of TAU contest		Compression rate	
	$\|N\|$	$\|E\|$	$\|N\|$	$\|E\|$
mgc_edit_dist	355111	498788	46.24%	55.74%
vga_lcd	2071117	2842994	49.70%	58.85%
leon3mp	2565434	3510170	49.53%	58.75%
netcard	307526	423687	52.90%	61.24%
leon2	2071117	2842994	46.46%	54.00%

$\|N\| = number of nodes on timing graph.$ $\|E\| = number of edges on timing graph.$

4.4 Macro Usage

To alleviate the design turnaround, the macro-models are frequently used in the inner loops of timing-driven optimization procedures. In fact, macro-models can be timed in both out-of-context usage (isolated) and in-context usage. Macro-models have to handle a critical amount of incremental changes in the parent-level circuits. It suffices to experiment the in-context usage of a macro-model since the out-of-context usage can be covered by the in-context usage. We change input and output boundary timing over hundreds of operations in this experiment. The total runtime is shown in Figure 4.1. It can be observed that by using our macro-models, the total runtime on all benchmarks can be substantially reduced to one-third of the flat timing. To sum up, the above experiments have demonstrated the promising performance

of our algorithm in terms of accuracy, high compression rate, and effective macro-usage.



(a) vga_lcd (b) leon3mp

(c) netcard (d) leon2

Figure 4.1: Runtime of timing analysis: $\times$ is our new timing model and $*$ is the original flat circuit.

CHAPTER 5

CONCLUSIONS

Our algorithm, LibAbs, provides an efficient method to abstract timing of a design and generate an accurate macro-model with a high compression rate. The macro-models generated by LibAbs reproduce accurate timing with accuracy loss within 0.3 ps. Compared to the original timing analysis, the model size of our generated macro-model is 32% compared to the original timing analysis. According to our experimental results, LibAbs outperforms the top performers from the 2016 TAU contest.

REFERENCES

[1] A. J. Daga, L. Mize, S. Sripada, C. Wolff, and Q. Wu, "Automated timing model generation," in *Proceedings of the 39th Annual Design Automation Conference - DAC '02*, pp. 146-151, 2002.

[2] C. W. Moon, H. Kriplani, "Timing model extraction of hierarchical blocks by graph reduction," in *Proceedings of the 39th Annual Design Automation Conference - DAC '02*, pp. 152-157, 2002.

[3] S. V. Venkatesh, R. Palermo, M. Mortazavi, and K. A. Sakallah, "Timing abstraction of intellectual property blocks," in *Proceedings of Custom Integrated Circuit Conference - CICC '97*, pp. 99-102, 1997.

[4] TAU Contest 2016, [Online]. Available: https://sites.google.com/site/taucontest2016/. [Accessed: 21-Sep-2016].

[5] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," *2015 IEEE/ACM International Conference on Computer-Aided Design - ICCAD '15*, pp. 895-902, 2015.

[6] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering,* vol. 5, no. 1, pp. 46 - 55, 1998.

[7] Illinois Campus Cluster, [Online]. Available: https://campuscluster.illinois.edu/. [Accessed: 21-Sep-2016].