DREAMTime: Timing-Driven State-of-the-Art Placer using static net weighting and fast parasitic net-modelling

Jay Zhe-An Mok
ECE Department of UBC
Vancouver, Canada
jm4304@student.ubc.ca

***Abstract*— A state-of-the-art analytical placer that is built on top of an AI Framework (DREAMPlace) is enhanced to allow for timing-driven optimization. Static timing-driven optimization is done, where a static timing analysis tool (OpenTimer) is used to compute the pre-placement design's slacks, which are then used to weight the nets in the placement's wire-length cost function. The design is then timed again after placement. For a number of large benchmarks with ~1 million gates and nets, the total negative slack can be reduced by 25% and worst-negative slack by 53% for a given circuit, with negligible effect to runtime and minimal increase of around 3% to total net wirelength. The code is on the following github repository: github.com/liu2333hui/DREAMTime**

Keywords*— *Placement, Static Timing Analysis, AI Framework

I. Introduction

With the growing number of transistors in today's chips, the need for CAD tools to scale and deal with larger designs while still meeting stringent requirements on timing, performance, power and area is ever more necessary. Physical design is an important process in the design, and a "good" placement are key to closing timing constraints and reducing buffers during clock tree synthesis. To this end, this work builds on a state-of-the-art analytical placer (DREAMPlace) built upon an AI framework by embedding an static timing analyzer into the optimization flow. By including timing information in the placement, critical paths and the slacks can be improved, thereby reducing the timing closure procedure and reducing timing buffer insertion steps. The original authors of DREAMPlace have also demonstrated timing-driven placement, but their methodology is slightly different and will be discussed later in section 3. In short, This work makes the following contributions,

- Timing-driven placement of large benchmarks can be done with little change to the placement algorithm, only the net-weighting, proportional to timing slacks, needs to be adjusted. But tns and wns can be reduced by more than 40% for benchmarks with millions of gates.
- Static net weighting can be as effective as dynamic net weighting, which is what the original authors of DREAMPlace 4.0 have implemented. The benefits include reduced runtime because the costly STA algorithm is not called repeatedly during placement, at little cost to the final timing and wirelength quality.
- Using the star-model instead of Steiner trees to model multi-pin net wirelengths, which is what the original paper used, the quality of results with respect to timing is not only similar but the runtime to generate the parasitic SPEF file is greatly reduced by more than 100x.

The paper is organized as follows. Section 2 describes the background for the EDA tools used in the experiment, such as the placement, STA tool and Steiner-tree generator. Section 3 describes the methodology, namely how the tools are used together to do timing-driven placement. Section 4 shows the results on a set of large million-gate benchmarks, and section 5 discusses future directions and concluding remarks.

II. Background

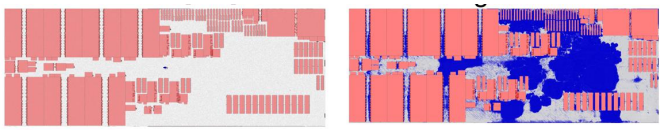

Figure 1. DREAMPlace Placement before (left) and after (right). The Red are macros, and blue are moveable cells.

A. DREAMPlace: State-of-the-art placer in AI Framework [1]

State-of-the-Art global placers are analytical due to the ability to converge to a fast solution that is close to the global optimum. DREAMPlace is one such placer where the optimization goal is to minimize the total net-wirelength and the "electro-static density" of a trial placement. Here, the density is a term borrowed from electro-static analysis and represents the total energy of the system if placement cells are treated as point charges, and . In short, lower net-length means less delay, less power, less area while lower density represents less overlap and congestion between cells (i.e. two cells too close, meaning two point charges are too close, will yield a high density energy). The optimization goal is thus [1],

$$\min_{\mathbf{w}} \sum_{i}^{n} \mathrm{WL}(e_i;\mathbf{w}) + \lambda D(\mathbf{w})$$

Where w is the placement cell locations, WL is a variant of the half-perimeter weire length, D is the density that was mentioned earlier, lambda is a heuristic weight.

Because the tool is built on top of an AI framework (PyTorch) in order to take advantage of optimized compute kernels and GPU resources, and because the optimization is done iteratively through gradient-descent techniques, the total

net-wirelength and density terms are differentiable. The exact equations can be found in the original DREAMPlace paper.

B. OpenTimer: An Open-source Static Timing Analysis [2]

Static timing analysis is the process of measuring the timing characteristics of a given design. Generally, timing libraries in the format of *liberty files* are read by the STA tool, which contains information such as gate arc delays depending on input transition times and output capacitance. Then, the STA tool, after reading in the netlist verilog, will perform a number of sweeps forward and backward to generate the required arrival times and arrival times of every net. If no SPEF file is given, the parasitics of nets are not considered and net delays are ignored. These arrival and required times then are subtracted to yield slacks, which can be used for critical path timing analysis, setup/hold violations, and other types of analysis. There will be multiple slacks based on the timing library corners (i.e. fast time, slow time, normal operation), and also on the analysis purpose (i.e. max for setup, min for hold).

C. Flute: Tool for generating Steiner trees for a net [3]

The key factor that differentiates placements with respect to timing analysis is the net delays. These are entirely placement dependent, but will need to be estimated due to the lack of routing information during placement. A common method is to use HPWL, clique-based, star-based or Steiner-tree for netlist wirelength estimation. In regards to Steiner-Tree based, the tool Flute, which uses look-up tables and net-breaking strategy, can be used to generate the Steiner-tree quickly for a given multi-pin net. Net wirelength is then a proxy for capacitance and resistances parasitics, which could then be written into a SPEF. Hence, Flute and other net wirelength estimators can be used to generate parasitic information as well.

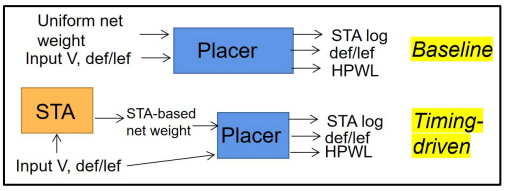

Figure 2. Overall methodology of timing-driven placement. The STA and RC Parasitic extraction (not shown but is used by the STA) is embedded in the optimization and will be used to update the net weights.

III. METHODOLOGY

The overall methodology is to embed the STA tool into the placer's optimization loop in order to yield better timing metrics. The overall flow-chart is shown in figure 2.

A. Timing-Driven Placement - Relation between placer and STA tool

Timing-driven placement involves adding timing information into the loop of optimization during placement. In theory, the STA tool should be called every iteration of placement and the timing information will update some part of the tool. In practice, due to the potential long runtime of STA tools (i.e. on a CPU 1-thread, can have runtimes of 1000s for a 2 million gate netlist), the tool will only be called once in a while. This is a heuristic and the work from [1] calls the STA every 500 sub-iterations, whereas this work will call the STA only once in the beginning.

B. Net Weighting Strategy Versus Path Based Strategies

There are two common ways to embed the timing information into a placer. One way is by weighting the nets in the HPWL sum differently depending on the net's *criticality*, a measure of how important the net is relative to timing goals. The criticality is a heuristic and can be defined in this way ,

$$crit. = min(1.0\,,1.0 + slack/wns)$$

In other words, STA slack information is implicitly included in the . This method is easy to implement, can scale to very large benchmark circuits, does not require a differentiable function for timing metrics such as WNS/TNS/Slack relative to the placement positions, and can yield good timing results as will be shown later.

Another method is to do path-based timing-driven placement. This includes finding the top-K% of critical paths and using those paths in the cost function. This is more costly but is more effective at reducing WNS.

Due to limited compute resources, and ease of implementation, the net weighting strategy is chosen.

C. Static Net Weighting Versus Dynamic Net Weighting

As mentioned previously, the STA generally cannot be called every iteration. Static net weighting only calls the STA once in the beginning and uses the initial floorplan's slack to update the net weights. Dynamic net weighting will update the weights continuously during placement. It will be shown later in the results section that dynamic net weighting from [1] gives better WNS and TNS, but at the cost of much higher runtime, whereas static net weighting can give similar HPWL quality with small to medium improvements in TNS/WNS with little runtime cost.

In this work, static net weighting is chosen due to its low cost on runtime.

D. Placement STA inputs and outputs, and RC parasitics

Unlike a post-routed design, which has all the information needed by a STA such as net delays, a pre-routed design only contains cell and macro placement information, without net parasitics or delays. Hence, in order to generate the slacks and timing metrics for a placed design (pre-routed), some method is required to estimate the net delays which are becoming significant in smaller technology nodes.

Parasitics are generated based on either the Flute-Steiner trees or the star-based tree. It was found empirically that using Steiner trees, although accurate, have high runtime up to 1 hour for 1 SPEF file on a single CPU, and running a SPEF using star-based parasitics can save huge runtime (a few seconds). This was also observed in [1] namely that SPEF generation is a bottleneck, almost half the runtime, and this is one way this bottleneck can be alleviated.

RC-parasitics are generated from the net models by multiplying netlist length against C or R per unit, and adding a PI-model RC network into the SPEF file.

IV. Results

The timing-driven placement is written in Python, with the STA, RC net generation tools coded in C++ but run through system calls in Python. The tools were compiled onto a laptop computer with an Intel 7 with 4 cores and 8GB RAM, running Ubuntu.The benchmarks were downloaded from the ICCAD15 timing-driven placement competition and the sizes of each circuit are shown below in table 1. Due to compute resource limitations (CPU with 8 GB Ram), the largest benchmarks superblue10 and 7 could not be run on my laptop.

TABLE I. Benchmarks for timing-driven placement. Cell and net counts are in the millions

case name	#cells	#nets	#pins	#rows
superblue1	1209716	1215710	3767494	1829
superblue3	1213253	1224979	3905321	1840
superblue4	795645	802513	2497940	1840
superblue5	1086888	1100825	3246878	2528
superblue7	1931639	1933945	6372094	3163
superblue10	1876103	1898119	5560506	3437
superblue16	981559	999902	3013268	1788
superblue18	768068	771542	2559143	1788

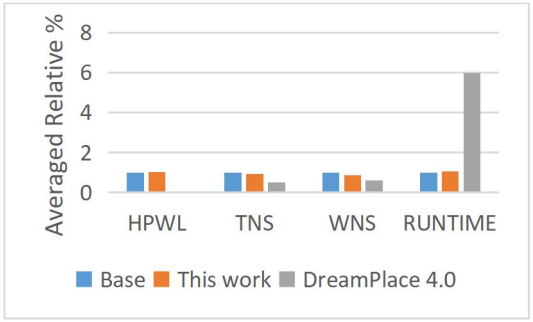

Figure 3. Different final placement normalized metrics such as HPWL, total negative slack (TNS), worst negative slack (WNS) and runtime are shown for different versions of placers.

The other benchmarks however were run and it is clear that the total negative slack and worst negative slacks can be improved by embedding timing information into the placement's cost function.

TABLE II. Relative quality of results between ref [1] and this work across different benchmark circuits

	HPWL	TNS	WNS	RUNTIME
Base	1	1	1	1
This work	1.0287	0.92758	0.8530	1.04029
DreamPlace 4.0 [1]	N/A	0.4918	0.6133	5.9842

It is clear from table 2 that adding timing-driven net weights reduces the WNS and TNS. Although the reduction is not as much as dynamic net weighting, the quality of HPWL is not increased by much and runtime is only increased by 4% compared with 600% with dynamic net weighting.

It is also interesting from table 3 that for some benchmarks, WNS is lower but TNS is higher, and vice versa. This is because static net weighting does not consider dynamic changes in the placement (net delays will definitely change during placement), and hence, it is possible for both optimization goals to not be fully reached. One way to improve upon this is to use dynamic net weighting, to use random placement seeds, or to vary the placement cost functions and how net weighting is done (i.e. can be scaled).

Future work includes exploring larger benchmarks, tuning placement hyper-parameters, and exploring how timing-driven placement can be sped-up by AI methods.

References

[1] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin and B. Yu, "DREAMPlace 4.0: Timing-driven Global Placement with Momentum-based Net Weighting," *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Antwerp, Belgium, 2022, pp. 939-944

[2] T. -W. Huang, C. -X. Lin and M. D. F. Wong, "OpenTimer v2: A Parallel Incremental Timing Analysis Engine," in *IEEE Design & Test*, vol. 38, no. 2, pp. 62-68, April 2021

[3] C. Chu and Y. -C. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, pp. 70-83, Jan. 2008

Appendix

The raw tables including all the placement runs, with and without timing-driven enabled, and the wirelengths are included below.

Table III. Full runtime, HPWL, TNS, and WNS data for baseline versus timing-driven placer.

	No timing-driven cost					With Timing-driven cost			
	Nets	Scaled HPWL (Ku)	Tns(ps)	Wns(ps)	Runtime (s)	Scaled HPWL (Ku)	Tns(ps)	Wns (ps)	Runtime (s)
Superblue1	1215710	4.24E+08	-28900600	-385206	3106	4.27E+08	-30406340	-309140	3194.836
Superblue3	951756	4.72E+08	-83709900	-439413	3405	4.73E+08	-64588200	-370077	3593
SUperblue4	802513	3.03E+08	-30097200	-309140	900	3.13E+08	-38426400	-260778	1001
Superblue5	1100825	4.63E+08	-66525400	-971069	1794.317	4.73E+08	-53610700	-561709	1794
Superblue16	756386	4.08E+08	-217094000	-208964	993	4.16E+08	-164458000	-319765	1072
Superblue18	771542	2.22E+08	-147031000	-580594	833	2.29E+08	-138113000	-281368	835
Simple	8	191	-28.5831	-14.2916	0.942	208	-25.4377	-12.7188	0.946

Code details

The major code changes made were in the following python files: “DREAMTimer.py”, “DREAMVerilog.py” and “PlaceDB.py”.

DREAMTimer.py includes functions that interfaces with the OpenTimer tool, generates the “cleaned” sdc file that is readable by the current version of OpenTimer 2.0 (i.e. it does not accept the command set_driving_cell, must convert it into an set_input_transition command), writes the appropriate OpenTimer script commands for getting the correct slack (because we care about tns and wns, it should be the slack from the maximum, or slow timing library), and parses the OpenTimer results for WNS/TNS/net slacks and converts it into the net weights used by DREAMPlace global placer.

DREAMVerilog.py includes functions that interfaces with the Flute tool, can create a SPEF file for RC parasitics that will be used by the OpenTimer tool for net delay estimation through Elmore Delay estimates, can use either star-based or Steiner-tree based net models (or maybe even based on simply fan-out) as the SPEF connections, and performs analysis of the input verilog to capture vital information such as placement of pins, net2pins names and other information which may be required by the STA tool.

PlaceDB is the main class that performs the placement optimization routine. This was modified in order to incorporate the timing-driven net weighting initialization step and the final timing check after placement. A dynamic net weighting scheme would also call DREAMTimer inside the inner-loop of placement as the net weights would then be changed dynamically throughout placement.

In the case where the OpenTimer has no slack information for a given net (i.e. can be from 10-30% of the time) due to false paths or other reasons, the net weight was assumed to be 1.0. Default values of the DREAMPlace algorithm were used (8.0 = gamma , 512 bins for solving Poisson’s equation for getting the Density term in the cost function). Future work could tune these hyper-parameters as the effect can be great.

Testing of the code was done manually (output a .sdc and .shell script for OpenTimer, then check the logs), or for Flute or star-based models, to plot the RC tree and calculate some sample delays.

README from the Github Repository:

Timing-Driven Placement , similar to work done by [1]P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin and B. Yu, "DREAMPlace 4.0: Timing-driven Global Placement with Momentum-based Net Weighting," 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2022, pp. 939-944.

To use-it, OpenTimer should be installed (the static timing analyzer tool). https://github.com/OpenTimer/OpenTimer/tree/master/example Furthermore, DREAMPlace should be compiled according to the original work. https://github.com/limbo018/DREAMPlace

Flute is included in the thirdparty folder as well as the detailed placer. Sample json hyper-parameters are in benchmarks.

Benchmarks were downloaded from the open-source ICCAD15 timing-driven placement circuits: http://iccad-contest.org/2015/problem_C/default.html#BENCHMARKS

The main added python scripts for timing-driven placement are : DREAMTimer.py and DREAMVerilog.py , both under dreamplace. The Placer.py is also modified slightly and PlaceDB.py (which has the main calls and flow for optimization).

DREAMTimer.py interfaces with the OpenTimer STA tool, and also "cleans" the sdc files. Outputs are also read and converted into appropriate weights here.

DREAMVerilog.py interfaces with the Flute tool, and also has an internal Star-based generator for RC parasitics. This script is mainly for generating the SPEF parasitic file, reading the verilog to get net2pin detailed mappings.

The PlaceDB.py "**call**" function is also modified to load in the weights from the timing analysis.