

A Novel Timing-Driven Placement Algorithm Using Smooth Timing Analysis

Andrey Ayupov, Leonid Kraginskiy

Strategic CAD Labs, Intel Corp.

andrey.ayupov@intel.com, leonid.kraginskiy@intel.com

Abstract

This work proposes a timing-driven placement algorithm that uses a new type of timing analysis, which we call smooth timing analysis. It constructs the timing cost function as a smooth function of cell placement. In addition, for net modeling the algorithm uses a companion net routing that provides more accurate wire delay. The placement task is then formulated as a non-linear optimization problem. Experiments prove that the proposed method is applicable to build timing-driven placement solutions for designs with thousands critical cells. Experimental results on blocks from recent microprocessor designs show a 65% average improvement in total negative slack comparing to a leading industrial flow.

1. Introduction

Currently there is great interest in new placement algorithms for cell-based designs. It is influenced by current technological trends and a transition to submicron process which results in new effects in interconnects, such as large resistance, noise, etc. The most common cost-function which is optimized in the basic placement algorithms is the total length of interconnects (or total wire-length, TWL). Indirectly, this helps to optimize timing of a circuit. However, it is necessary to take into account quadratic dependency of wire delay on wire length, different cell characteristics (load capacitance, drive strength), and path criticality to achieve satisfactory results for industrial high-performance designs.

This work proposes a method to introduce accurate timing models in the core of the placement algorithm. We show a technique that enhances the internal STA algorithm to calculate a TNS function of cell coordinates which can be used in placement optimization. We use global routing to model interconnects in placement, which makes timing estimation more accurate. We have chosen to use analytic placement as a core of our algorithm, as it is very convenient for formulating non-linear objective function, and to use regularization idea from [1] for smoothing cusps in functions, such as absolute and maximum value.

The rest of the article is organized as follows: in section 2 we describe the placement problem

formulation and its cost function. In section 3 the idea and implementation of dealing with timing cost is described. Section 4 will cover some optimization strategies. In section 5 we present the placement flow and experimental results. Section 6 includes the final remarks and conclusions.

2. Placement problem formulation

We use an analytical placement engine similar to APlace [2]. APlace is formulated as an unconstrained non-linear optimization task minimizing the cost function consisting of two terms: wirelength and cell spreading. In this work the placement task is formulated as an unconstrained non-linear problem minimizing the following cost function:

$$Cost = CellPenalty + SmoothTiming \quad (1)$$

where *CellPenalty* is a spreading term that was borrowed from [2] and *SmoothTiming* estimates timing quality of a circuit which will be defined further in this section.

We use the quasi-Newtonian method from the TAO toolkit [5] to minimize cost function (1). In this method it is required that the value of the cost function and its partial derivatives with respect to all independent variables (the gradient vector) be calculated at every optimization iteration.

The *SmoothTiming* cost term approximates the total negative slack (TNS) of the circuit:

$$SmoothTiming = \sum_{e_i \in E} \max(0, -slack(e_i)) \quad (2)$$

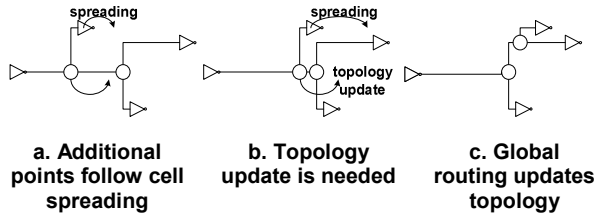
where the calculation of $slack(e_i)$ is described in the following section, and E is the set of endpoints of combinational logic, i.e., primary outputs and input pins of sequential elements.

For timing optimization it is important to model interconnects in the placement accurately. We propose to use global routing to model nets during placement. We then allow the routes to change during placement optimization. The net is represented by a Steiner tree with additional (Steiner) points that are used for topology structuring. The idea of the approach is to consider those additional points as movable during placement optimization. The coordinates of those points are treated as independent optimization variables along with the cell coordinates. During cell spreading, the Steiner tree may also require updating (Fig. 1). The frequency of routing updates using global router is defined by the user.

Thus, every net consists of a set of two terminal segments. A terminal may be either a cell pin or a Steiner point. For a segment with two terminals which are positioned at (x_1, y_1) and (x_2, y_2) , we estimate its length using Manhattan distance:

$$\text{SegmentLength} = |x_1 - x_2| + |y_1 - y_2| \quad (3)$$

All cell positions and Steiner points in the design constitute a set of independent variables V . Let us denote the space of all V values as X .



3. Smooth timing analysis

In this section we describe how we evaluate the value and gradient vector for $\text{slack}(e_i)$. Our regular static timing analysis (STA) algorithm takes a placed and routed design as an input and then estimates the wire R/C as a linear R/C multiplied by the estimated length of a wire segment (3). The flow of calculations in STA for a given placement solution $\bar{x} \in X$ is shown in the Fig. 2.

routing segments \rightarrow wire lengths \rightarrow wire R/C \rightarrow
loads at each point \rightarrow cell and wire delays \rightarrow
arrival and required times (AT/RT) \rightarrow slacks

Figure 2. STA calculation flow

All functions in this flow such as routing segment lengths, delays, arrival/required times and slacks are functions of \bar{x} . The basic idea of our approach is to make these values actual functions and to operate on functions in the same way as it was done for floating values.

Having a function $O(a, b)$, operating on values a and b , we can construct an operator $O(f, g) = h$ where $h(\bar{x}) = O(f(\bar{x}), g(\bar{x}))$. For example, for the wire delay computation $\text{Wire_Delay}(\text{segment_length}, \text{segment_load}) = r \cdot \text{segment_length} \cdot \text{segment_load}$, we can construct corresponding operator on functions of a vector $\bar{x} \in X$:

$\text{Wire_Delay}(\text{segment_length}, \text{segment_load}) = f$,
where

$$f(\bar{x}) = r \cdot \text{segment_length}(\bar{x}) \cdot \text{segment_load}(\bar{x}).$$

Using the formula of the derivative of complex function (the chain rule):

$$(\nabla O)(f, g) = O'_f \cdot \nabla f + O'_g \cdot \nabla g, \quad (4)$$

one can see that for the value and gradient vector calculation of operator $O(f, g)$ at a given point $\bar{x} \in X$, only the value and gradient vector of functions $f(\bar{x})$ and $g(\bar{x})$ are needed and there is no need for their

symbolic forms. Having this in mind, we find it useful to introduce a helper structure *PFunc* which represents a function with only the value and gradient vector stored.

Let F_f be an instance of type *PFunc* and f be a function that F_f represents at the particular point \bar{x} . Then F_f has a container which stores a value $f(\bar{x})$ and all partial derivatives $\partial f / \partial x_k(\bar{x})$ at this point. Let us use the following notation:

$$F_f = (f(\bar{x}), [\partial f / \partial x_1, \partial f / \partial x_2, \dots, \partial f / \partial x_n]).$$

The following example illustrates the idea.

Example: For simplicity, let us assume that a function of two variables x and y is to be minimized. Then $F_x = (v_1, [1, 0])$ and $F_y = (v_2, [0, 1])$ represent functions $f_1(x, y) = x$ and $f_2(x, y) = y$ at the point $(x, y) = (v_1, v_2)$. Let $f(x, y) = x^2 y$ and F_f be representations at the point $(x, y) = (v_1, v_2)$. Using F_x, F_y , representations of the independent variables stated above, one can express $F_f = F_x^2 \cdot F_y$ as the composition of two operators: power and multiplication. Then, the power operator produces $F_x^2 = (v_1^2, [2v_1, 0])$ and multiplication yields final result:

$$F_f = (v_1^2 v_2, [2v_1 v_2, v_1^2]).$$

In this example, we need to implement the value and gradient calculation for the multiplication and power operators on *PFunc* structures. The pseudo-code in Fig. 3 gives an example of multiplication operator implementation. Once the value and gradient calculation have been implemented for the basic mathematical operators as $*$, $/$, power, etc for arguments F_x and F_y , one can compute a smooth approximation of any complex function we come across in this paper.

```

PFunc operator Mult (PFunc p1, PFunc p2) {
  PFunc res;
  res.value = p1.value * p2.value;
  for  $i$  from 0 to size of  $X$  {
    res.grad[i] = p1.grad[i] * p2.value +
                 p2.grad[i] * p1.value
  }
  return res;
}

```

Figure 3. Pseudo-code of operator Mult

For non-smooth operators like abs , max and min , we use smooth calculation of values and partial derivatives based on *logsumexp* approximation from [3], i.e.:

$$\min(x, y) = -\eta \cdot \log(\exp(-x/\eta) + \exp(-y/\eta)) \quad (5)$$

$$\max(x, y) = \eta \cdot \log(\exp(x/\eta) + \exp(y/\eta)) \quad (6)$$

$$|x| = \eta \cdot \log(1 + \exp(x/\eta)) + \eta \cdot \log(1 + \exp(-x/\eta)) \quad (7)$$

where η is a parameter to control the accuracy of approximation.

We have extended the regular STA implementation so that it can use the new *PFunc* structure instead of floating point values. Thus, it calculates wire loads, RC delays, cell delays, arrival/required times, and slacks in

a recursive manner using (4) as functions of independent variables V . Finally, we calculate value and gradient of *SmoothTiming* cost term according to definition (2).

Having implemented STA operating on *PFunc*, we get a new type of analysis which produces partial derivatives along with the value calculation for the timing cost function. Additionally, it helps to smooth cusps of non-smooth functions. Therefore, we call it “smooth timing analysis”. Using this idea it is possible to convert any analysis algorithm into an optimization task and use all the accurate models of the analysis algorithm in the optimization.

5. Optimization

Our static timing analysis algorithm performs worst-case max-timing analysis. For cell delays (slopes, setups) we use linear forms:

$$\begin{aligned} \text{delay}_{in \rightarrow out} &= d^0 + d^1 \cdot \text{load}_{out} + d^2 \cdot \text{slope}_{in} \\ \text{slope}_{out} &= s^0 + s^1 \cdot \text{load}_{out} + s^2 \cdot \text{slope}_{in} \\ \text{setup}_{in \rightarrow clk} &= c^0 + c^1 \cdot \text{load}_{out} + c^2 \cdot \text{slope}_{in} + c^3 \cdot \text{slope}_{clk} \end{aligned} \quad (8)$$

For wires we use PERI [4] (close to Elmore) delay-model with slope propagation. RC extraction is done using a π -model with segment R and C proportional to segment length with linear wire R and C, individual for each layer.

Since we calculate *SmoothTiming* at each optimization iteration it is required the smooth timing analysis to be fast enough. Initially, smooth analysis was much slower than standard STA. How much slower depends on a benchmark (path lengths, fanout). Another major issue is memory consumption. Now we present the runtime improvement techniques which we have used and their impact on the quality of results and runtime performance. These improvements lead to an algorithm with acceptable runtime and memory consumption for designs with critical subcircuits of several thousands of cells.

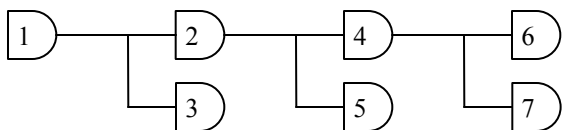


Figure 4. Slope propagation

Within our delay model only positions of the cells in the direct fanout of the driver influence its delay through the driver load calculations. Delay dependence

on the input slope is more complex. If we calculate the delay and slope completely throughout the path (Fig. 4) as in formulas (9), the delay will depend on positions of all preceding cells in the path and their fanout.

$$\begin{aligned} \text{delay}_4 &= d^0 + d^1 \cdot \text{load}_4 + d^2 \cdot \text{slope}_{4in} \\ \text{load}_4 &= f(x_4, y_4, x_6, y_6, x_7, y_7) \\ \text{slope}_{4in} &= g(\text{slope}_2, x_2, y_2, x_4, y_4, x_5, y_5) \\ \text{slope}_2 &= s^0 + s^1 \cdot \text{load}_2 + s^2 \cdot \text{slope}_{2in} \end{aligned} \quad (9)$$

This results in both long runtime and large memory consumption and can not be used in practice for large circuits. If we reduce slope calculation in formula (9) to $\text{slope} = s^0 + s^1 \cdot \text{load}$, this will give a satisfactory estimation of the slope and reduce dependence of the cost function to the direct neighborhood only.

Another major runtime improvement is to reduce the dependence of arrival times on slowest paths. Imagine that we have a 2-input gate. We then calculate arrival time at the output as $AT = \max(AT_1, AT_2)$ where AT_1 and AT_2 are arrival times through different pins. Then if AT_1 and AT_2 are close values we should take the dependence on both inputs. But if AT_1 is significantly larger than AT_2 then the dependence of the resulting arrival time on AT_2 and thus on the whole fanin cone of the second input could be neglected. This optimization is achieved naturally with the following technique: when we store function derivatives, we store only non-zero values in a vector (sparse array). To implement it, we occasionally clean up small insignificant values of derivatives in order to reduce dependence, save memory and avoid unnecessary additions of zero and multiplications by zero.

An important feature of our STA algorithm is its incremental mode. Using this mode we run STA on the critical subcircuit only, which is selected as shown in section 6. In this case arrival/required times for inputs/outputs of the subcircuit are constants, which are obtained from regular STA run from the previous iteration. In the Table 1 we present timing values, algorithm runtime and memory demand for different optimization modes. The experiment was performed on Intel® Xeon™ 2.80GHz machine with 4Gb of memory.

6. Experimental results

We have implemented our placement algorithm with a timing-driven optimization term in C++ on Linux. A quasi-Newtonian method from TAO toolkit [5] is used to solve unconstrained non-linear problems. For our experiments we used internal circuits from a recent

Table 1. Algorithm optimization statistics

15K Cells Critical - 2.5K	Real STA	Smooth STA Fair Models	Smooth + Slope Opt	Smooth + SMax Opt	Increm. Smooth Fair Models	Increm. + Slope Opt	Increm. + SMax Opt
WNS (ps)	-34	-	-32	-32	-37	-32	-32
TNS (ps)	-3206	-	-2421	-2421	-3747	-2634	-2634
Runtime (s)	3.6	Out Of	82	32	26	8.1	4.9
Memory (Mb)	2	Memory	2350	764	1015	307	201

Intel microprocessor designed for a 45nm technology. The proposed timing optimization is used as an incremental stage after a leading timing-driven industrial flow. In general, it could also be used in the main placement engine.

Our incremental placement flow consists of two phases. First, 50 iterations of purely timing-driven placement are run using the following cost function: $Cost = SmoothTiming$. Then the spreading term is added to the cost function: $Cost = CellPenalty + SmoothTiming$. It operates until nearly legal placement is produced. In our experiments, global placement ends when the average overlap between cells is less than 3%. Afterwards, sizing and placement legalization are run to produce legal results. These two stages were used at the end of the industrial flow as well.

We used the incremental mode of STA in timing optimization (see section 4) as it shows a good tradeoff between runtime and accuracy. The subcircuit was selected based on slack threshold provided by a user. The optimized subcircuit comprises cells that have output pins with slack less than a given threshold (critical subcircuit). In addition, the first level of fanin and fanout of the critical subcircuit has been also added to this subcircuit as cells in the fanin and fanout greatly affect the cell delays.

In the experiments two optimization flows were examined. For both flows the slack threshold was set to 10ps to control degradation of timing of nearly critical cells during placement.

For the first flow the critical subcircuit selection was done once in the beginning. In the second flow the subcircuit selection is updated every 25 placement iterations. Results of both flows are compared with the results of industrial flow in the Table 2.

For six industrial circuits the cell count is given in parentheses. The “A” columns reflect results of the leading industrial flow; the “B” columns correspond to the proposed flow without critical subcircuit update and the “C” columns correspond to the flow with critical subcircuit update. WNS and TNS are given in picoseconds; total wire length (TWL) and area are given as ratios to the industrial flow. The “Average” column shows TNS, TWL and area improvement/degradation of the two flows against the industrial flow. The data in the table shows that the proposed optimization flow could further improve the

results of the industrial flow by 65% in TNS and 1.4% in area. The congestion report of the internal global routing tool after the proposed optimization did not show any noticeable degradation of routability.

7. Conclusion

In this paper we proposed a novel algorithm for timing-driven placement using smooth timing analysis. Smooth timing analysis produces partial derivatives (gradient vector) along with value calculation for the timing cost function. This gradient vector is used to direct timing optimization in an analytical placement framework. The smooth timing analysis uses the same accurate delay models as regular STA. Routing trees were used in placement algorithm to model interconnects. We demonstrated applicability of proposed approach to designs from recent microprocessor. Experimental results showed that using the proposed timing optimization in placement after a leading industrial flow improves timing (TNS) of a circuit by 65% on average.

8. Acknowledgment

We would like to thank Alexander Marchenko, Andrey Zhmurin and Oleg Venger for their support and useful feedback.

9. References

- [1] R. Baldick, A.B. Kahng, A. Kennings and I.L. Markov, “Efficient Optimization by Modifying the Objective Function: Applications to Timing-Driven VLSI Layout”, *IEEE Trans. on Circuits and Systems - I: Fundamental Theory and Applications*, v. 48, no. 8, 2001, pp. 947-956.
- [2] [A.B. Kahng and Q. Wang, “Implementation and extensibility of an analytic placer”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, V. 24, Issue 5, 2005, pp. 734-747.
- [3] A. Ruehli, P. Wolff Sr. and G. Goetzl, “Analytical power timing optimization techniques for digital systems”, *Proc. of the 14th ACM/IEEE Design Automation Conference*, 1977, pp. 142-146.
- [4] C.V. Kashyap, C.J. Alpert, F. Liu and A. Devgan, “Closed Form Expressions for Extending Step Delay and Slew Metrics to Ramp Inputs”, *Proc. of Int. Symp. on Physical Design*, 2003, pp. 24-31.
- [5] S.J. Benson, L.C. McInnes, J. More and J. Sarich, “TAO User Manual”, <http://www.mcs.anl.gov/tao>.

Table 2. Experimental results

	Circuit (cell count)															Average				
	Ckt1 (2537)			Ckt2 (8069)			Ckt3 (15450)			Ckt4 (9995)			Ckt5 (12836)					Ckt6 (21094)		
Flow	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	A	B	C	B	C
TWL	1.00	1.05	1.04	1.00	1.03	1.02	1.00	1.02	1.01	1.00	1.03	1.03	1.00	1.01	1.01	1.00	1.00	1.00	-2.4%	-1.8%
Area	1.00	0.98	0.99	1.00	0.99	1.00	1.00	0.98	0.98	1.00	0.99	0.98	1.00	1.00	0.97	1.00	1.00	1.00	1.1%	1.4%
WNS	-11	-1	-1	-31	-38	-32	-8	-5	-5	-29	-18	-18	-21	-4	-2	-31	-31	-31	-	-
TNS	-18	-1	-1	-834	-740	-635	-38	-12	-7	-268	-213	-188	-311	-11	-6	-3568	-1290	-1425	59.5%	65.0%