Techniques for Effective Distributed Physical Synthesis

Freddy Y.C. Mang
Synopsys, Inc.
fmang@synopsys.com

Wenting Hou
Synopsys, Inc.
wthou@synopsys.com

Pei-Hsin Ho
Synopsys, Inc.
pho@synopsys.com

ABSTRACT
We present two techniques, (1) placement-based timing-driven partitioner (PTP) and (2) virtual physical synthesis based budgeter (VSB), that support effective distributed physical synthesis.

Categories and Subject Descriptors
J.6 [**Computer-aided engineering**]

General Terms
Algorithms, Design, Theory.

Keywords
Distributed physical synthesis, partitioning, time budgeting.

1. INTRODUCTION
Physical synthesis performs placement and placement based logic optimization to the design, which is expensive in terms of both CPU and memory usage and has become the runtime and capacity bottleneck of the IC implementation flow. An obvious solution for reducing the runtime and memory usage of physical synthesis is divide-and-conquer; i.e., automatically partitioning the design into blocks and distributing the computation for these blocks over a network of servers. Traditionally this divide-and-conquer process is manually carried out by engineers using a floorplanner that requires intensive user intervention and often incurs great impact to the timing of the design due to suboptimal partitioning and time budgeting. To enable effective distributed physical synthesis, we present two techniques, (1) placement-based timing-driven partitioner (PTP) and (2) virtual physical synthesis based budgeter (VSB) that support fully automatic distributed physical synthesis and minimize the impact of divide-and-conquer to design timing.

Placement-based timing-driven partitioner (PTP) utilizes a placement of the design to estimate the timing of and the proximities between the cells to make partitioning decisions, while conventional partitioners are based on the netlist not the placement of the design. In comparison, PTP greatly reduces the impact of partitioning to timing. On 7 industrial designs PTP improved design timing by 24.4% on average. PTP does not support distributed initial placement but supports distributed placement-based logic optimization. With modern placement techniques [3][7][11], placement based logic optimization, not placement itself, is the runtime and capacity bottleneck of physical synthesis, so we believe that PTP provides a good trade off between the performance of the design and the performance of distributed physical synthesis for designs that can be effectively handled by a modern placer.

DAC 2007, June 4–8, 2007, San Diego, California, USA

To enable distributed placement based logic optimization, we need timing and load constraints for the ports of the blocks of the partition. For example, in Figure 1, we partitioned the design into two blocks on the left and right. To optimize the two blocks concurrently, timing and load constraints are required for both sides of the port *P*. Conventional time budgeters allocate times that are proportional to the current delays of the timing paths on both sides of the port. For example, suppose that the clock period is *9 ns* and the current delays of the timing paths *AP* and *PB* are *10* and *5 ns* respectively. Then a conventional time budgeter would assign the budgets of *6* and *3 ns* to the timing paths *AP* and *PB* respectively. But it may turn out that the delay of the timing path *AP* can be easily reduced to *4 n*s, but to save cell area and power the optimization engine stopped at *6 ns* once the timing constraint of *6 ns* is met. The delay of timing path *PB* may stay at *5 ns* because no trick can be done to reduce the delay. As a result, when we merge the optimized left and right blocks at the end of distributed physical synthesis, the delay of the timing path *AB* becomes *11=6+5 ns* and violates the timing constraint of *9 ns* by *2 ns*. If the budgeter had assigned *4* and *5 ns* to the timing paths *AP* and *PB* respectively, the delay of the merged timing path *AB* would have become *9 ns* and satisfied the timing constraint.

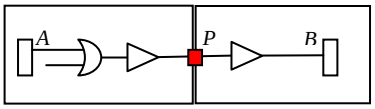

Figure 1 Timing path from the flop *A* to flop *B* is broken by the port *P* after the design is partitioned into two blocks.

Therefore, ideally a time budgeter should consider the "potential for timing optimization" during time budgeting. For example, the potentials for timing optimization of the timing paths *AP* and *PB* are *6* and *0 ns* respectively. Virtual physical synthesis based budgeter (VSB) employs *virtual physical synthesis (VPS)* to estimate the potential for timing optimization. On 7 industrial designs, VSB improved design timing by 23.4% on average.

The rest of the paper is organized as follows. We present PTP and the related work in Section 2. In Section 3 we describe VSB and the related work. Section 4 shows the experimental results of PTP and VSB and Section 5 concludes the paper.

2. Placement-Based Timing-Driven Partitioning (PTP)
Large industrial designs are hierarchical; i.e., the top-level design instantiates some logical modules and each logical module may instantiate some other logical modules. To facilitate formal

equivalence checking and ECO (Engineering Change Order), industrial partitioners respect the logical module boundaries as much as possible. Figure 2 shows an example of a design before and after partitioning by an industrial partitioner. On the left is the logical hierarchy of the design before partitioning, in which each circle represents a logical module and rectangle represents a standard cell. The root, the level-1 node, of the tree, represents the top-level design. A logical module *M* is the parent of another logical module or standard cell *N* if logical module *M* instantiates the logical module or standard cell *N*. Only the first three levels of the logical hierarchy of the design are shown in the figure.

On the right is the physical hierarchy of the design after partitioning. Each level-*2* circle represents a block of the partition. Again, only the top 3 levels of the physical hierarchy are shown.

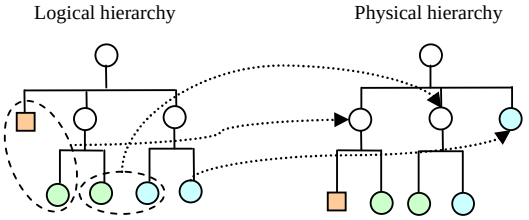

Figure 2 Partitioning introduces physical hierarchy

After partitioning, conventional floorplanners assign each block a non-overlapping rectilinear area in the floorplan. All cells of each block will be placed and optimized inside the assigned rectilinear area, including the new cells introduced by logic optimization. A placement before partitioning can be used to guide the division of the floorplan into the rectilinear areas. The disadvantage of this *rectilinear-block approach* is that the rigid block boundaries may require cells to be moved away from their ideal locations and thus lead to suboptimal design timing. The advantage is that there would be no overlapping cells after merging the block-level results of distributed physical synthesis.

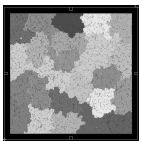

Figure 3 Ameba blocks for distributed physical synthesis

Alternatively, based on a placement of the entire design, we can assign each block a non-exclusive "ameba-shaped" area. Figure 3 is an example of this *ameba-block approach*. The advantage of this approach is that the flexible block boundaries may lead to better design timing. The disadvantage of this approach is that due to the potentially overlapped block areas, we must legalize the placement after merging the distributed physical synthesis results, which may lead to timing degradation. PTP and VSB support both the rectilinear-block and ameba-block approaches. We used the ameba-block approach in our experimental flow to obtain the experimental results.

2.1 PTP high-level algorithm

At a high level, we solve the partitioning problem using a clustering algorithm. First, the user specifies the ideal number *k* of blocks that the partition should contain. Based on the total number *n* of standard and hard-macro cells of the design, we compute the average number $m = \lfloor n/k \rfloor$ of cells in each block of the partition. Let *u* be the total number of cells with negative slacks after placement-based timing estimation. We also compute the average number of $v = \lfloor u/k \rfloor$ of *negative-slack cells* in each block of the partition.

Second, we create a *clustering priority queue* by traversing the logical hierarchy tree starting from the root, the top-level design, in a breadth-first traversal. Let ε be a constant between *0* and *1* (we pick $\varepsilon = 0.07$ for all experiments). Whenever we encounter a logical module, if it contains more than $(1-\varepsilon)\cdot m$ cells or $(1-\varepsilon)\cdot v$ negative-slack cells, we continue traversing all of its children. Otherwise we do not traverse beyond this logical module. When the traversal terminates, each logical module and standard cell at which we stopped the traversal becomes a *cluster* in the initial clustering priority queue.

Third, we perform a greedy best-first clustering algorithm on the clustering priority queue. We iteratively delete the two clusters whose merger provides the highest *clustering gain* (to be defined later) and each contains fewer than $\lfloor n/(k-1) \rfloor$ cells and $\lfloor u/(k-1) \rfloor$ negative-slack cells from the clustering priority queue and insert the union of the two clusters back into the clustering priority queue, until we have between *k-1* and *k+1* candidates left in the clustering priority queue or there are no clusters that can be merged. Then each remaining cluster of the clustering priority queue becomes a block of the partition. In subsequent sections, we define the clustering gain to encourage the blocks to assume more integral areas in the placement and reduce the number of inter-block critical timing paths.

2.2 Closeness indicator

The PTP clustering algorithm is based on an initial placement of the design. In the partition we prefer each block to occupy a more integral rather than dispersed area in the initial placement, since an integral block area implies (1) less total cell displacement from the initial placement for the rectilinear-block approach and (2) less total cell displacement from the block-level results of distributed physical synthesis for the ameba-block approach. Less total cell displacement usually means better design timing.

To achieve more integral block areas, we want to cluster clusters whose cells are mingled or close in the initial placement. We define a *closeness indicator* that measures how mingled and close the cells of two clusters are. Denote by *ab(C)* the area of the bounding box of the cluster *C*. We define the closeness indicator *closeness(C,D)* for two clusters *C* and *D* as follows.

$$closeness\ (C, D) = \frac{ab(C) + ab(D)}{ab(C \cup D)} - 1$$

The closeness indicator is a number between *-1* and *1*. The more overlapped the two clusters are the greater the number is. We use the two examples in Figure 4 to illustrate how to calculate the closeness indicator. For the two clusters on the left and right, the closeness indicator *closeness(C,D)* are (12+9)/30-1 = -0.3 and (12+9)/20-1 = 0.05 respectively.

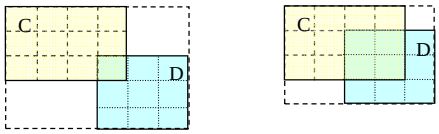

Figure 4 Closeness indicator

2.3 Criticality indicator

Even with an ideal time and load budgeting, optimizing a segment of a timing path in separation may reduce applicable optimization tricks and thus may negatively impact the timing of the entire timing path. Therefore we want to include the entire critical timing path inside a block of the partition as much as possible.

Given two candidate clusters *C* and *D* in the clustering priority queue, we call a port a *canceled interface port of the clusters C and D* if the port is only at the boundaries of *C* and *D*; i.e., properly contained in the union of *C* and *D*. Denote by *ci_port(C,D)* the set of canceled internal ports of the clusters *C* and *D*. Denote by *WNS* the worst negative slack of the entire design and *WNS(p)* the worst negative slack of all timing paths through the port *p*. To achieve the goal of making a critical timing paths "intra-block", we define the *criticality indicator* for two clustering candidates *C* and *D* as follows.

$$criticality(C,D) = \max\left\{\frac{WNS(p)}{WNS} : p \in ci\_port(C,D)\right\}$$

The timing indicator is a number between *0* and *1*. The more critical a timing path is between the two clustering candidates the greater the criticality indicator is.

2.4 Clustering gain

The clustering gain of two clusters is a linear combination of the closeness and the criticality indicators of the two clusters. Given a constant coefficient α between *0* and *1*, the linear combination *gain(C,D)* is defined as follows.

$$gain(C,D) = \alpha \cdot closeness\ (C,D) + (1-\alpha)\cdot criticality(C,D)$$

The smaller the constant α is the larger the influence of timing criticality is to the greedy clustering algorithm in Section 2.1. We used the constant $\alpha = 0.2$ for all of our experiments, which gave the best timing results comparing to other constants that we tried.

2.5 PTP related work

Classic netlist-based partitioning methods like Kernighan-Lin [10], Fiduccia-Mattheyses [5] and hMetis [9] do not consider the logical hierarchy, placement or timing of the design. The method DHML [4] considers the logical hierarchy but not placement or timing. Placement-based clustering algorithms discussed in [2] and [6] do not consider logical hierarchy or timing. Since considering logical hierarchy is a key prerequisite for industrial partitioners we compare PTP against DHML in our experiments.

3. Virtual physical synthesis budgeter (VSB)

To concurrently optimize each block of the partition we need time and load budgets for the ports of each block before distributed physical synthesis. In Section 1 we use Figure 1 to illustrate the importance of considering the "potential of timing optimization" during time budgeting. Virtual physical synthesis based budgeter (VSB) does that by (Step 1) quickly estimating the delay of the design after physical synthesis by *virtual physical synthesis (VPS)* and (Step 2) generating the budgets based on the time and load estimated by VPS. In this section we first introduce the *proportional time budgeting algorithm* that is used by both VSB (based on the estimated timing) and a conventional budgeter (based on the current timing); i.e., Step 2 of the VSB algorithm. Then we introduce the *virtual physical synthesis (VPS)* technique; i.e., Step 1 of the VSB algorithm.

3.1 Proportional time budgeting algorithm

We again use the timing path in Figure 1 to illustrate the proportional budgeting algorithm. Suppose that the worst slack $r - a$ at the port *P* is given by the arrival time *a* and the required time *r*. Note that the arrival time *a* is the delay from the startpoint of the worst timing path to the port *P*. Let *c* be the required time at the endpoint of the same worst timing path through the port *P*. Then $c - r$ is the delay from the port *P* to the endpoint of the same worst timing path. The proportional time budgeting algorithm assigns the input delay of $a \cdot c/(a + c - r)$ to the port *P* if *P* is an input port of the block, or an output delay of $(c - r) \cdot c/(a + c - r)$ to the port *P* if *P* is an output port of the block.

3.2 Virtual physical synthesis (VPS)

Virtual physical synthesis (VPS) takes in a placed netlist and estimates the timing of the design after physical synthesis without actually modifying the placement or the netlist. VPS does that using the analytical models that we developed for the most commonly used placement-based logic optimization tricks: (1) buffer chain insertion, (2) gate sizing and (3) repeater insertion. In the subsequent subsections we describe the modeling for each one of the above three tricks in detail.

3.2.1 Buffer Chain Insertion

To drive a large load, physical synthesis inserts a buffer chain in front of the original driver. We call the total delay of a buffer chain the *insertion delay* of the buffer chain. We assume that the library contains buffers of so many different sizes that the buffers are approximately continuously sizable. Note that the purpose of VPS is for VSB, not for really optimizing the design, so we believe that making the above assumption is not unreasonable.

For a buffer *B* driving a load *L* with input transition time *t*, let $transition_B(t,L)$ and $delay_B(t,L)$ be the output transition time at the output of the buffer *B* and the delay through *B* respectively. We also define the *gain g* of the buffer as L/C, where *C* is the input capacitance of the buffer *B*. According to the logical effort theory [12], to minimize insertion delay, each buffer in a buffer chain should have the same gain *g*. We model a buffer chain as a buffer tree that (1) is rooted at the same leftmost buffer as the buffer chain, (2) has *g* times as many buffers in the next level as in the current level, and (3) has the same number of levels as the buffer chain (see Figure 5).

We can see that the buffer tree has the same insertion delay as the original buffer chain. In addition, each level of the buffer tree has the same delay, so the insertion delay of the buffer tree is the delay of one level multiplied by the number of levels. We have

thus reduced the buffer chain insertion problem to the following problem: given a load L, we want to compute the buffer B and the level N for the buffer tree rooted at the buffer B such that the insertion delay of the buffer tree is minimized.

For a buffer B with input capacitance C and a gain g, we select a transition delay t such that both the input and output transition times of B equal to t, i.e., $t = transition_B(t, gC)$. For a buffer chain driving a load L, the number of levels N of buffers required is $N = \log(L/C)/\log(g)$, and the insertion delay is therefore $insDelay_B(L, g) = N \cdot delay_B(t, gC)$.

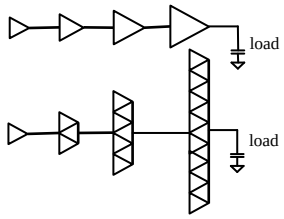

Figure 5 Buffer chain modeled as buffer tree with gain = 2

Since the delay function of the buffer B is convex in the parameter g, the insertion delay function $insDelay_B$ is also convex in g. Hence a minimum value for the insertion delay exists. We iterate through all the buffers in the buffer library and choose the buffer B^* that gives the minimum insertion delay and let g^* be the associated gain. Note that the buffer B^* and gain g^* are independent of the size of the load L, and as a result, we can use the same buffer and gain for all different loads.

We pick the nominal transition delay t^* so that $t^* = transition_{B^*}(t^*, g^* C^*)$. This nominal transition delay will be used for modeling the gate sizing and repeater insertion tricks.

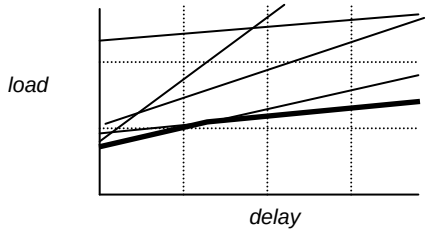

Figure 6 Load-delay model for all two-input AND cells. There are 5 different cell sizes. The function $Delay_{\min}(load)$ is depicted as the thick bilinear function at the bottom.

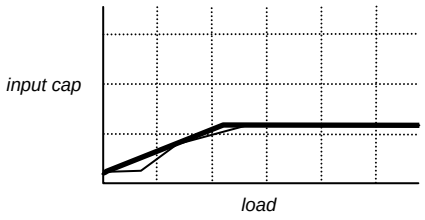

Figure 7 Load-capacitance model for all two-input AND.

3.2.2 Gate Sizing

We use an analytical model to estimate the impact of gate sizing to delay. We assume that the input transition delay is the nominal input transition delay t^* calculated in Section 3.2.1. We also assume that all input pins of a cell are symmetric; i.e., they are of the same input-to-output delay and capacitance. Then for each class of cells of the same logic function f in the library, say, all 2-input AND cells, we compute two piecewise linear functions: (1) given a load, the *load-delay function* returns the minimum delay for the logic function f and (2) the *load-capacitance* function returns the input capacitance of the cell that realizes the minimum delay. Note that in the logical effort theory [12], both of these two functions are linear. Our analytical model can be considered as a generalization of the logical effort theory. Now we show how to compute the load-delay function. For each cell of the same logical function f, we plot the load-delay curve as in Figure 6. Then for each output load, we record the minimum delay among all cells of the same logical function f and fit the data points by a bilinear function as follows.

$$Delay_{\min}(load) = \begin{cases} a \times load + b & if\ load \le e \\ c \times load + d & if\ load > e \end{cases}$$

The load-capacitance function can be similarly computed. For each output load, we record the input capacitance of the cell that realizes the minimum delay and again fit the data points by a piecewise linear function (see Figure 7 for an example):

$$inCap(load) = \begin{cases} C_{\min} & if\ load \le c1 \\ a \times load + b & if\ c1 < load \le c2 \\ C_{\max} & if\ load > c2 \end{cases}$$

3.2.3 Macro-aware Repeater Insertion

It is a well-known fact that the delay of an un-buffered wire is quadratic to the length of the wire. Physical synthesis inserts repeaters (buffers or inverters) to the wire and makes the wire delay linear in length. In VPS, we first choose a repeater for the repeater insertion. We assume that repeaters are inserted at equal distance in the wire..

Given a repeater B with output resistance R, input capacitance C, and inter-repeater distance l, the inter-repeater delay is:

$$d = R(c \cdot l + C) + rl(c \cdot l / 2 + C) + delay_B(t^*, c \cdot l + C),$$

where r and c are the wire resistance and capacitance per unit length respectively, t^* is the nominal delay computed in Section 3.2.1, and $delay_B(t, load)$ is the delay of the repeater B given input transition time t and output load $load$.

For a long wire of length L, the delay $D(L, l, B) = d \cdot L / l$ is simply the number of repeaters times the inter-repeater delay. Since the delay function is convex in l, a minimum delay $D(L, l^*, B)$ exists, where l^* is the optimal inter-repeater distance. Since the optimal delay $D(L, l^*, B)$ is linear in L, *we can* then iterate through the buffers and inverters in the library to search for the repeater B^* such that the delay per unit wire length $D^*(L, l^*, B^*)/L$ is minimum.

In practice, the repeaters along a long wire should be of progressively increasing sizes. We assume that each successive repeater is *h* times larger in area than the previous one (Figure 8). The first repeater in the repeater chain is the optimal repeater *B**. For a long wire of length *L* and a load *load* at the sink, $k = L/l^* - 1$ repeaters are inserted, and we assign $h = \sqrt[k]{load/C}$. Assuming that the repeater that is *h* times larger than *B** has an input capacitance $C \cdot h$ and output resistance *R/h*, the delay for the *i-th* repeater-to-repeater segment is:

$$d_i = R/h^i(cl^* + h^i C) + rl(cl^*/2 + h^i C) + delay_{B^*}(t^*, cl + h^i C)).$$

The delay for the buffered wire is therefore $\sum_{0 \le i \le k} d_i$.

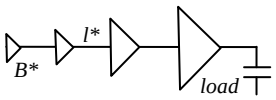

Figure 8 Repeaters of progressively increasing sizes are inserted at equal distance into a long wire.

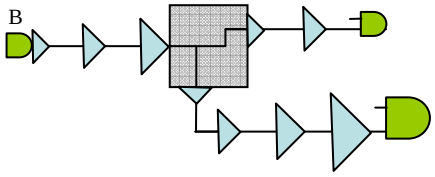

Figure 9 Example of macro-aware repeater insertion. Repeaters of increasing sizes are inserted right before and after the macro, and at equal distance on the segments of the net that are not blocked.

3.3 VPS high-level algorithm

Given a placed design and a cell library, we first estimate the input capacitances of the cells in design. This is done by backward traversal, starting from the timing endpoints. For a cell, we first obtain an estimate of the load. Then the input capacitance of this cell is obtained from its load-capacitance model in Section 3.2.2. The load of a cell is given by the input capacitance of the buffer used for buffer chain insertion if buffer chain insertion was done, or that of the optimal repeater if repeaters were inserted to the driven net, or simply the current total load of the cell, i.e. sum of the capacitance of the net and the load of the sink pins.

For short nets in the design, we assume that the net delay is negligible. If the current total load of the driving cell of the net is larger than the input capacitance of the buffer chosen in Section 3.2.1, we employ buffer chain insertion to drive the load. We assume that the delays from the source pin to all the sink pins are the same for multi-pin nets.

For a long net in the design, we first generate a Steiner route for the net. If the route goes through a large hard macro where placement of repeaters is prohibited, we add the pre-calculated repeater *B** obtained in Section 3.2.3 after the hard macro. This models what physical synthesis would typically do in the presence of hard macros. For segments of the net that are not inside the hard macros, repeaters of increasing sizes are added at optimal inter-repeater distances. Finally, we calculate the delay from the driver of the net to a sink by adding up the Elmore delay of different segments of the path. An example is given in Figure 9.

VSB performs load budgeting as follows: for an output port *P* of a block, we assign a load budget equal to the estimated load seen by the driving cell of the net *N* to which the port *P* belongs. For an input port *P*, we set an input resistance equal to the estimated output resistance of the driving cell, which is RC/C', where *R* and *C* are the current output resistance and average input capacitance of the current driving cell, and *C'* is the estimated input capacitance of the resized driving cell.

3.4 VPS experimental results

We compare VPS with a commercial physical synthesis tool on 13 industrial designs. For each design, we compare the worst negative slacks (WNS) predicted by VPS versus the result of the actual physical synthesis. Prediction error is the absolute difference between the WNS as a fraction of the clock period. We also compare the runtimes. The results are shown in Table 1. On average VPS is 78x faster than the actual physical synthesis and within 11% error in WNS.

Table 1 VPS vs. physical synthesis: WNS and runtime

	#cells	Clock Period (ns)	Original WNS (ns)	VPS WNS (ns)	P-SYN WNS (ns)	Error (%)	Speedup (X)
D1	22K	10	-107.48	0.79	-2.93	37.26	193
D2	23K	11	-6.31	-6.18	-5.10	9.81	207
D3	24K	5.2	-15.93	-7.42	-6.01	26.97	149
D4	57K	288	-4.23	-0.01	-0.46	0.15	62
D5	65K	10	-881.24	0.43	0.00	4.33	32
D6	86K	3.63	-3.14	-0.28	-0.13	4.16	59
D7	161K	5.8	-8.82	-3.51	-2.37	19.69	69
D8	292K	5.4	-8.77	-1.01	-1.98	17.97	34
D9	330K	200	-8.43	-1.57	-2.21	0.32	30
D10	395K	21.6	-14.83	-8.19	-8.61	1.96	31
D11	396K	13	-86.02	-2.55	-3.01	3.54	51
D12	504K	60	-4.40	-1.53	-2.65	1.87	18
D13	613K	100	-285.43	-264.35	-268.29	3.94	72
Average						**10.15**	**78**

3.5 VSB related work

The authors in [8] described a method to estimate the minimum achievable delay of a design. Like VPS, their method is based on a form of logical effort model for the cell library. However, they only consider the gate-sizing trick and do not consider the placement of the cells nor wire delay. Our repeater insertion technique was inspired by [1]. However, their choice of repeaters is based on previous runs of similar designs on the same technology, and they only consider one type of repeaters.

4. Experimental Results

We implemented both placement-based timing-driven partitioner (PTP) and virtual physical synthesis based budgeter (VSB) on the top of a state-of-the-art commercial physical synthesis tool. To measure the timing improvements that PTP and VSB can achieve, we developed an experimental physical synthesis flow that: (1) perform timing-driven placement for the entire design, (2) run PTP, (3) run VSB, (4) run distributed placement-based logic optimization using the ameba-block approach, (5) merge the block-level results and legalize, and (6) quick touch-up to fix electrical DRC violations caused by legalization. We ran the

experimental physical synthesis flow on 7 industrial designs with between 300K and 2.5M cells on a network of 64-bit AMD servers with 2.2GHz CPUs and 16G main memories.

Table 2 shows the comparison between the experimental distributed physical synthesis flow and the standard physical synthesis (flat) flow. The first three columns show the design code names, the cell numbers and the number of blocks in the partitions of the designs. The last two columns show the runtime speedup and timing degradation. On average the experimental flow achieved a 1.7x runtime speedup with 4.8% timing degradation. The range of timing degradation is less than ideal. The main purpose of this experimental distributed physical synthesis flow is to measure the capabilities of PTP and VSB. We believe that this experimental flow can be refined to achieve more runtime speedup and less timing degradation.

Table 2 Experimental distributed synthesis flow vs. flat flow

Design name	# cells	# blocks	Speedup	WNS comp. (negative=improved)
V	305K	9	1.4x	4.80%
T	629K	8	1.6x	4.10%
H	850K	3	1.4x	-9.10%
W	1.1M	8	2.2x	1.20%
L	730K	8	1.7x	9.70%
A	1.3M	5	1.5x	15.40%
S	2.4M	7	2.1x	7.30%
Average			**1.7x**	**4.8%**

Table 3 compares the partitioning methods PTP and DHML [4] by running the step (2) of the experimental flow with "run PTP" vs. "run DHML". On average the experimental flow with PTP spent 1.2% more CPU time but achieved 24.4% better timing. For designs W and A, both PTP and DHML quickly decided to use the level-2 logical modules as the blocks of the partitions. Table 4 compares the budgeting methods VSB and the proportional time budgeter based on the current design timing by running the step (3) of the experimental flow with "run VSB" vs. "run proportional time budgeter". On average the experimental flow with VSB spent 1.2% more CPU time but achieved 23.4% better timing.

Table 3 PTP vs. DHML

Design Name	CPU time (negative = improved)	WNS (negative = improved)
V	-3.78%	-19.47%
T	5.27%	-6.72%
H	8.55%	-58.74%
W	0.00%	0.00%
L	-1.79%	-84.24%
A	0.00%	0.00%
S	-2.50%	-1.69%
Average	**1.15%**	**-24.41%**

5. Conclusions and Future Work

Distributed physical synthesis improves runtime and capacity, but may incur negative impact to timing. We introduced placement-based timing-driven partitioner (PTP) and virtual physical synthesis based budgeter (VSB) to enable high-quality distributed physical synthesis. PTP considers logical hierarchy, placement and timing during partitioning and VSB performs a fast virtual physical synthesis algorithm to produce more realizable time and load budgets. On 7 real-world designs an experimental distributed physical synthesis flow with PTP and VSB on average achieved respectively 24.4% and 23.4% timing improvements with negligible runtime overhead. Both PTP and VSB are also useful techniques in conventional floorplanners. Indeed, VSB has been productized in a commercial floorplanner.

Table 4 VSB vs. proportional time budgeter

Design Name	CPU time (negative = improved)	WNS (negative = improved)
V	0.07%	-9.00%
T	10.98%	-3.10%
H	6.94%	-47.53%
W	-3.62%	-27.27%
L	-4.30%	-71.41%
A	9.50%	-7.19%
S	-11.49%	1.75%
Average	**1.16%**	**-23.39%**

6. References

[1] C.J. Alpert, J. Hu, S.S. Sapatnekar and C.N. Sze, Accurate estimation of global buffer delay within a floorplan. ICCAD, pp. 706-711, 2004.

[2] H. Chen, C.-K. Cheng, N.-C. Chou and A.B. Kahng, An Algebraic Multigrid Solver for Analytical Placement with Layout Based Clustering. DAC, pp. 794-799, 2003.

[3] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, A High-Quality Mixed-Size Analytical Placer Considering Preplaced Blocks and Density Constraints, ICCAD, pp. 187-192, 2006.

[4] Y. Cheon and D.F. Wong, Design Hierarchy Guided Multilevel Circuit Partitioning. TCAD, Volume 22, Issue 4, pp. 420- 427, 2003.

[5] C.M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions. DAC, pp. 175-181, 1982.

[6] B. Hu and M. Marek-Sadowska, Wire Length Prediction based Clustering and its Application in Placement.

[7] A.B. Kahng, S. Reda, and Q. Wang. Aplace: A general analytic placement framework. ISPD, pp. 233–235, 2005.

[8] S.K. Karandikar and S.S. Sapatnekar, Fast comparisons of circuit implementations. IEEE Trans. VLSI Syst. 13(12), 05.

[9] G. Karypis, R. Aggarwal, V. Kumar and S. Sheckhar, Multilevel hypergraph partitioning: Application in VLSI domain. DAC, pp. 526-529, 1997.

[10] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning of electrical circuits. Bell System Technical Journal, vol. 49, no. 2, pp. 291-307, 1970.

[11] P. Spindler and F.M. Johannes. Fast and Robust Quadratic Placement Combined with an Exact Linear Net Model. ICCAD, pp. 179-186, 2006.

[12] I. Sutherland, B. Sproull and D. Harris. Logical effort: designing fast CMOS circuits. Morgan Kaufmann Publishers Inc., 1999.