Block-level 3D IC Design with Through-Silicon-Via Planning

Dae Hyun Kim[1], Rasit Onur Topaloglu[2], and Sung Kyu Lim[1]
[1]Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332
[2]GLOBALFOUNDRIES
Email: daehyun@gatech.edu, rasit.topaloglu@globalfoundries.com, limsk@ece.gatech.edu

***Abstract*— Since re-designing and re-optimizing existing logic, memory, and IP blocks in a 3D fashion significantly increases design cost, near-term three-dimensional integrated circuit (3D IC) design will focus on reusing existing 2D blocks. One way to reuse 2D blocks in the 3D IC design is to first perform 3D floorplanning, insert signal through-silicon vias (TSVs) for 3D inter-block connections, and then route the blocks. In this paper, we propose algorithms (finding signal TSV locations, assigning TSVs to whitespace blocks, and manipulating whitespace blocks) for post-floorplanning signal TSV planning in the block-level 3D IC design. Experimental results show that our signal TSV planner outperforms the state-of-the-art TSV-aware 3D floorplanner by 7% to 38% with respect to wirelength. In addition, our multiple TSV insertion algorithm outperforms a single TSV insertion algorithm by 27% to 37%.**

I. Introduction

As 2D ICs are designed at various design levels such as block level and gate level, 3D ICs can also be designed at various design levels. In the core-level 3D IC design, we put existing 2D IC layouts together, insert signal, power/ground, thermal, and dummy TSVs, fabricate each die, and stack and bond the dies. The primary merit of the core-level design is that we can fully utilize 2D CAD tools to design each die and reuse highly-optimized 2D IC layouts.

In the block-level 3D IC design, we perform 3D floorplanning with existing 2D blocks, insert TSVs into whitespace, fabricate each die, and stack and bond the dies. The primary merit of the block-level design is that we can reuse existing highly-optimized blocks without major modification. Since re-designing and re-optimizing each block in a 3D fashion is very costly, using existing well-designed blocks is inevitable in the 3D IC design.

In the gate-level 3D IC design, we flatten the whole design, place gates and TSVs in 3D, fabricate each die, and stack and bond the dies. Since the gate-level 3D IC design provides the highest degree of freedom on gate and TSV locations, previous works focus on the gate-level 3D IC design. However, re-designing a whole circuit in the gate-level 3D IC design significantly increases design cost. In addition, pre-bond testing is also becoming a serious overhead in this design level [1].

One of the most important issues in the 3D IC design is that locations of signal TSVs have a huge impact on the design quality. Ill-placed signal TSVs cause long detours, so the performance of 3D ICs having poorly-placed TSVs could be worse than that of 2D ICs. Therefore, we should take signal TSV locations into account in the 3D IC design. While many papers address signal TSV insertion in the core-level and the gate-level 3D ICs [2]–[5], few work inserts signal TSVs physically in the block-level 3D IC design [6]–[8]. In addition, some of these block-level 3D IC design works do not use realistic wirelength metrics, so they significantly underestimate total wirelength. Furthermore, they do not consider multiple signal TSV insertion, which is essential for wirelength minimization.

In this paper, we propose algorithms for signal TSV planning in the block-level 3D IC design. Our contributions are as follows:

This material is based upon the work supported by the National Science Foundation under Grant No. CCF-0917000, Semiconductor Research Corporation (SRC ICSS), and the Interconnect Focus Center (IFC).

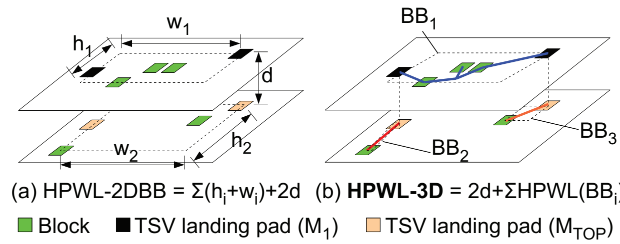

Fig. 1. Wirelength metrics for a 3D net. (a) HPWL based on 2D bounding boxes. (b) HPWL based on subnet construction. d is the vertical length of a TSV.

- We propose a more accurate wirelength metric for use in the block-level 3D IC design.
- We propose a post-floorplanning signal TSV insertion method for the block-level 3D IC design.
- We develop an effective algorithm for 3D rectilinear Steiner tree construction to find 3D routing topologies.
- We develop a multiple signal TSV insertion algorithm for wirelength minimization.

To the best of our knowledge, this is the first work on block-level signal TSV planning that uses a more realistic wirelength metric and incorporates multiple signal TSV insertion.

II. 3D Wirelength Metrics

In this section, we review 3D wirelength metrics and propose a more accurate wirelength metric for use in the multiple TSV insertion. The following terminologies distinguish two signal TSV insertion methods.

- **Single TSV insertion**: To connect blocks placed in two adjacent dies, we use only one TSV.
- **Multiple TSV insertion**: To connect blocks placed in two adjacent dies, we use multiple TSVs if inserting multiple TSVs reduces the total wirelength further.

A. 3D Half-Perimeter Wirelength Based on Bounding Boxes

One simple way to compute the wirelength of a 3D net is to construct a *3D bounding box* containing blocks and TSVs in the 3D net and sum the width, the height, and the vertical length of the 3D bounding box. We call this wirelength metric *HPWL-3DBB* (HPWL based on a 3D bounding box). [6], [7] use this wirelength metric. However, HPWL-3DBB significantly underestimates the wirelength.

Another way to compute the wirelength of a 3D net is to construct *2D bounding boxes* containing blocks and TSVs in each die in the 3D net. After 2D bounding box construction in each die, we sum the HPWL of each 2D bounding box and the vertical length of a TSV multiplied by the number of TSVs. We call this wirelength metric *HWPL-2DBB* (HPWL based on 2D bounding boxes). Fig. 1(a) shows an example of HPWL-2DBB. If we use the single TSV insertion, HPWL-2DBB produces the most accurate HWPL-based 3D wirelength.

978-1-4673-0772-7/12/$31.00 ©2012 IEEE

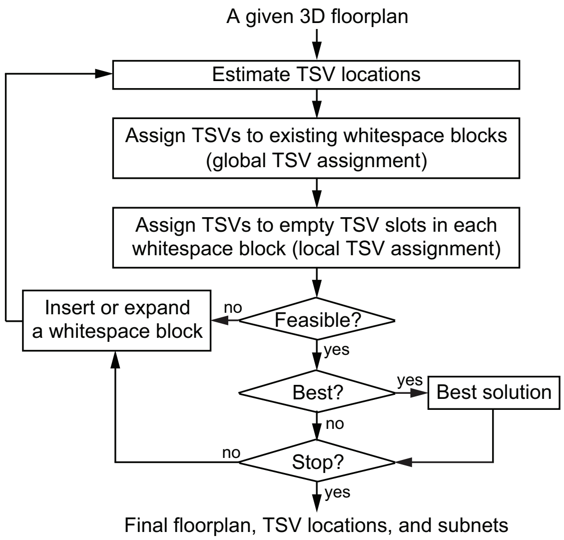

Fig. 2. Our signal TSV planning flow.

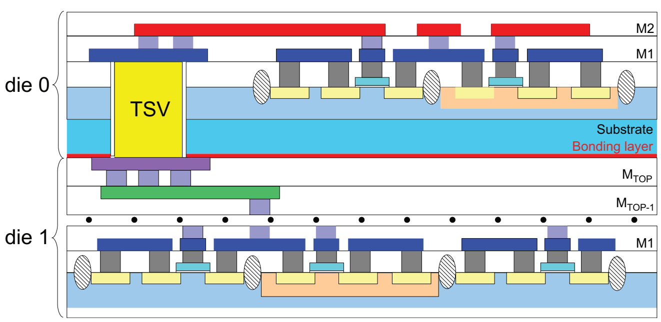

Fig. 3. A 3D IC with via-first TSV and face-to-back die stacking.

B. Subnet-based 3D Half-Perimeter Wirelength

If we use the multiple TSV insertion, HPWL-2DBB computes the wirelength of a 3D net inaccurately. In fact, the multiple TSV insertion splits a 3D net into multiple subnets as shown in Fig. 1(b). In this case, each subnet has its own bounding box, so we can compute the total wirelength of a 3D net H_i more accurately as follows:

$$\text{HPWL-3D}(H_i) = d \cdot N_{\text{TSV},i} + \sum \text{HPWL}(\text{BB}_{i,j}) \ , \quad (1)$$

where d is the vertical length of a TSV, $N_{\text{TSV},i}$ is the total number of TSVs used for net H_i, and $\text{HPWL}(\text{BB}_{i,j})$ is the HPWL of the 2D bounding box of the j-th subnet of H_i. HPWL-3D also computes the wirelength of the single TSV insertion accurately.

III. Signal TSV Planning

Fig. 2 shows our signal TSV planning flow for the block-level 3D IC design. Since we focus on post-floorplanning steps, we assume that 3D floorplans are given to us. For a given 3D floorplan, we first find TSV locations minimizing wirelength regardless of locations of available whitespace. To find TSV locations, we construct a 3D rectilinear Steiner tree (RST) for each 3D net, apply a bottom-up breadth-first search to the 3D RST to find a die span (defined in Section IV) of each Steiner point, and determine TSV locations. We show our algorithms for the construction of a 3D RST and the determination of TSV locations in Section IV.

In general, floorplanners generate compact floorplans, so TSV locations found by algorithms ignoring available whitespace locations are likely to be located on functional blocks. Since we cannot insert TSVs into functional blocks, we should find available whitespace close to estimated TSV locations. We solve this problem by TSV assignment, which we explain in Section V.

If we fail to assign TSVs to whitespace due to lack of enough whitespace, we insert a new whitespace block, expand an existing whitespace block, or redistribute whitespace blocks. Since this whitespace manipulation changes the given floorplan, if we change the current floorplan, we go back to the step for estimation of TSV locations as shown in Fig. 2. We present our whitespace manipulation algorithm in Section VI.

In this work, we assume that we use via-first TSVs and face-to-back die stacking as illustrated in Fig. 3.

IV. Estimation of TSV Locations

2D rectilinear Steiner minimum tree (RSMT) construction algorithms are frequently used to find optimal routing topologies for 2D nets. Similarly, since we can replace a planar (x- or y-directional) edge by a metal wire and a vertical (z-directional) edge by a TSV, we can use 3D RSMT construction algorithms to find optimal routing topologies for 3D nets. However, there is no published work on 3D RSMT construction. In this section, therefore, we develop a 3D RST construction algorithm using a 2D RSMT construction algorithm to find TSV locations as well as 3D routing topologies. Fig. 4 briefly illustrates our 3D RST construction algorithm. In Fig. 4(a), a 3D net has six pins to be connected. In Fig. 4(b), we project these points onto a 2D plane. In Fig. 4(c), we construct a 2D RSMT for the projected points. To construct a 2D RSMT, we use FLUTE [9]. In Fig. 4(d), we expand the 2D RSMT to a 3D RST. When we expand a 2D RSMT to a 3D RST, some of the Steiner points in the 2D RSMT should connect multiple dies as shown in Fig. 4(d). Therefore, we compute a die span of each Steiner point during the 2D to 3D expansion. Here we define a *die span* as follows:

Definition 1: A ***die span*** of a point is the range of dies that the point connects.

For example, in Fig. 4(c) and Fig. 4(d), Steiner point s1 is supposed to connect p0 in die 0 and p2 in die 2, so the die span of s1 is $[0, 2]$.[1]

After 3D RST construction, we insert TSVs into and between Steiner points in the 3D RST and construct subnets. These TSV locations are used for estimated TSV locations in our signal TSV planning flow.

A. Computation of a Die Span of a Steiner Point

The set of points of a 2D RSMT consists of fixed points (i.e., input points) and Steiner points inserted by a 2D RSMT construction algorithm. In Fig. 4(c), for example, p0 to p5 are fixed points and s1 to s4 are Steiner points. When we expand a 2D RSMT to a 3D RST, we construct a 3D RST by inserting vertical edges at Steiner points as shown in Fig. 4(d). However, when we insert vertical edges into Steiner points, we should determine which dies each Steiner point connects. We solve this problem by computing a die span of each Steiner point.

To compute a die span of each Steiner point in a given 2D RSMT, we apply the bottom-up breath-first search algorithm to the 2D RSMT. In Fig. 4(c), for example, we first visit depth-0 points (p0 to p5), then depth-1 points (s1, s2, s3), and then depth-2 points (s4).[2] The reason that we apply the bottom-up breath-first search algorithm is because the computation of die spans of higher-depth Steiner points (e.g., depth-1 points) needs determined die spans of lower-depth points (e.g., depth-0 points) adjacent to them.

[1] Notice that the die number of the topmost die (die 0) is 0 while that of the bottommost die (die d-1) is d-1 where d is the number of dies.

[2] The *depth* of a point is defined as the minimum depth from the root point set (the set of fixed points).

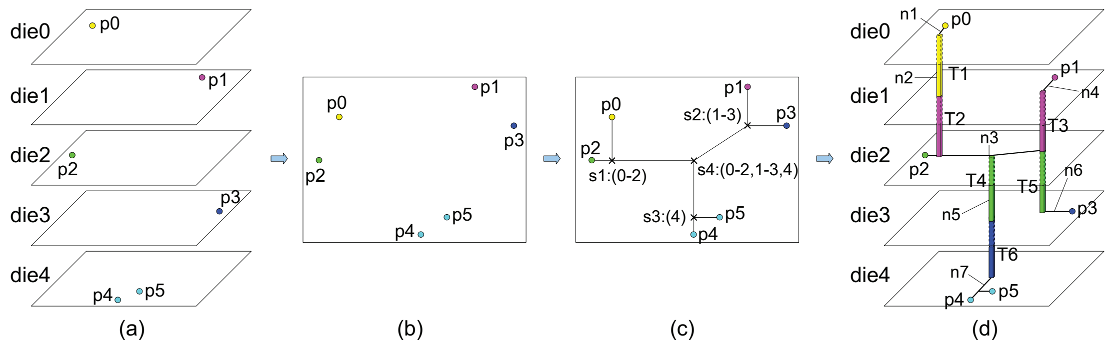

Fig. 4. Construction of a 3D RST. (a) Points to be connected. (b) Fixed points projected onto a 2D xy plane. (c) A 2D RSMT. (d) A 3D RST constructed from (c).

Algorithm 1: Our 3D RST construction algorithm.

Input: A set $F = \{p \mid p \in \mathbb{Z}^3\}$ of fixed 3D points.
Output: TSV locations and subnets.

1 $E \leftarrow$ **Construct_2D_RSMT** (F);
2 $Q \leftarrow \{\}$; // a queue.
3 **for** *each* $p \in F$ **do**
4 $\quad p$.visited $\leftarrow$ **true**;
5 $\quad p$.top $\leftarrow p$.die; $\quad p$.bot $\leftarrow p$.die;
6 $\quad Q$.enqueue (p);
7 **end**
8 **while** *!Q.empty()* **do**
9 $\quad p_1 \leftarrow Q$.dequeue();
10 $\quad$ **for** *each unvisited point* p_2 *adjacent to* p_1 **do**
11 $\quad\quad$ tTop $\leftarrow \infty$; $\quad$ tBot $\leftarrow -\infty$;
12 $\quad\quad$ **for** *each visited point* p_3 *adjacent to* p_2 **do**
13 $\quad\quad\quad$ tTop $\leftarrow$ **MIN** (p_3.bot, tTop);
14 $\quad\quad\quad$ tBot $\leftarrow$ **MAX** (p_3.top, tBot);
15 $\quad\quad$ **end**
16 $\quad\quad$ **if** *tTop* $>$ *tBot* **then**
17 $\quad\quad\quad$ tTop $\leftarrow$ **IRand** (tBot, tTop);
18 $\quad\quad\quad$ tBot $\leftarrow$ tTop;
19 $\quad\quad$ **end**
20 $\quad\quad p_2$.top $\leftarrow$ tTop; $\quad p_2$.bot $\leftarrow$ tBot;
21 $\quad\quad p_2$.visited $\leftarrow$ **true**;
22 $\quad\quad$ **for** *each unvisited point* p_3 *adjacent to* p_2 *s.t.* $p_3 \notin Q$ **do**
23 $\quad\quad\quad Q$.enqueue (p_3);
24 $\quad\quad$ **end**
25 $\quad$ **end**
26 **end**

Algorithm 1 shows our algorithm for the computation of a die span at each Steiner point during the 2D RSMT to 3D RST expansion. We first create an empty queue, Q (Line 2). Then, for each fixed point p in F, we set its *visited* variable to true (Line 4), which denotes that this point is visited and this point has a fixed die span. We also set its *top* and *bot* variables to its die number (Line 5). For example, if a point p is located in die1 ($p.die$=1), its *top* and *bot* become 1. The *top* and *bot* variables denote the topmost die and the bottommost die that the point connects, respectively. We then insert these points into Q (Line 6) for the breath-first search.

Between Line 8 and Line 26, we apply the breath-first search algorithm. First, we dequeue a point p_1, which is a point whose die span is already computed, from Q (Line 9). Then, we compute a die span of each unvisited point p_2 adjacent to p_1.[3] For this, we prepare two temporary variables, *tTop* and *tBot*, and initialize them (Line 11). Then, for each visited point p_3[4] adjacent to p_2, we set *tTop* to the smaller number of p_3*.bot* and *tTop* (Line 14) and set *tBot* to the larger number of p_3*.top* and *tBot* (Line 15). This computation finds the minimal die span, which connects all the visited points adjacent to p_2, of p_2. For example, in Fig. 4(c), we first visit p0. Since s1 is an unvisited point adjacent to p0, we compute a die span of s1 by visiting all visited points (p0 and p2) adjacent to s1. Then, the die span of s1 becomes $[0, 2]$ by the computation in Line 12 to Line 15 in Algorithm 1.

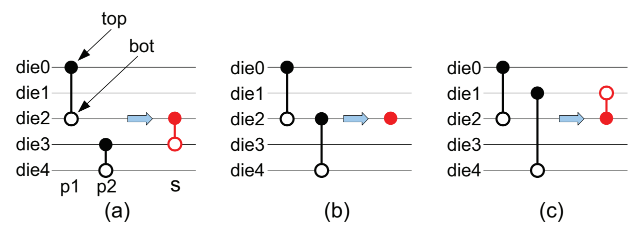

Fig. 5. Die span diagrams. Solid dots are *top* variables and empty dots are *bot* variables. Red spans show *tTop* and *tBot* when we determine the die span of s. (a) *tTop* (=2) $<$ *tBot* (=3). (b) *tTop* (=2) $=$ *tBot* (=2). (c) *tTop* (=2) $>$ *tBot* (=1).

When we compute the die span at Steiner point $p2$, three relations between *tTop* and *tBot* can exist as illustrated in Fig. 5. If *tTop* is smaller than *tBot*, we need edge(s) connecting from *tTop*-th die to *tBot*-th die (Fig. 5(a)). If *tTop* equals *tBot*, we do not need vertical edges because we can use planar edges to connect visited points adjacent to s (Fig. 5(b)). If *tTop* is greater than *tBot* as shown in Fig. 5(c), there are overlaps among die spans of visited points adjacent to s, so we do not need to insert vertical edges. In this case, we just choose a die in $[tBot, tTop]$ to connect s and visited points adjacent to s in 2D (Line 16 to Line 19). The *IRand(a,b)* function in Line 17 returns an integer number in $[a, b]$.

Then, we set the die span of p2 (Line 20), mark p2 as a visited point (Line 21), and enqueue all unvisited points adjacent to p2 into Q (Line 23) for the breath-first search.

B. Insertion of TSVs into and between Steiner Points

After we expand a 2D RSMT to a 3D RST, we insert TSVs into and between Steiner points as follows. If *top* of a Steiner point is

[3] Unvisited points are always Steiner points.
[4] A visited point always has a determined die span.

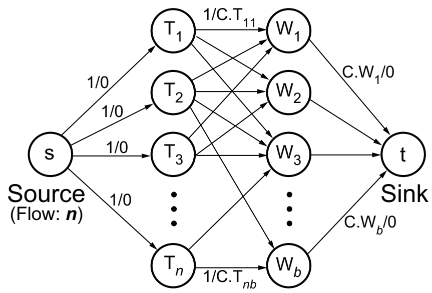

Fig. 6. Global assignment of TSVs to whitespace blocks. T_i is the i-th TSV and W_j is the j-th whitespace block. f/c in each edge denotes that f is the maximum flow capacity, and c is the cost. $C.T_ij$ is the wirelength when TSV T_i is assigned to whitespace block W_j. $C.W_i$ is the maximum number of available TSV slots in whitespace block W_i.

smaller than its *bot*,[5] we insert TSVs from the (*top*)-th die to the (*bot*-1)-th die.[6] This is an insertion of TSVs into a Steiner point.

If the die spans of two adjacent Steiner points do not overlap, we also insert TSVs between the two Steiner points. For example, if the die span of a Steiner point s1 is $[1, 2]$ and the die span of a Steiner point s2 adjacent to s1 is $[4, 5]$, we need to insert a TSV in die 2 and a TSV in die 3 between these two Steiner points. In this case, we insert TSV(s) in the middle of the two points.

C. Construction of Subnets

After we find TSV locations for a 3D net, we construct subnets for the net. For instance, the net in Fig. 4(d) consists of the following subnets: $n1$ connecting p0 and the metal 1 landing pad of TSV T_1, $n2$ connecting the bottom landing pad of TSV T_1 and the metal 1 landing pad of TSV T_2, $n3$ connecting p2, the bottom landing pads of TSV T_2 and T_3, and the metal 1 landing pads of TSV T_4 and T_5, and so on.

Our subnet construction algorithm is based on iterative search. For a point p in a 3D RST, we create an empty set S, insert p into S, and traverse adjacent points from p. If an adjacent point j is in the same die with p, we insert j into S. If j is in a different die, we stop traversing through j. In this case, j is in the upper die, so we add the bottom landing pad of j into S. After we finish traversing, we find a non-empty set S, which becomes a subnet. We repeat this process until we traverse all the points in the 3D RST.

V. TSV Assignment

Since TSVs cannot be inserted into functional blocks, we should assign estimated TSV locations to nearby whitespace blocks, as illustrated in Fig. 2. To assign TSVs to whitespace blocks, we use a minimum-cost flow formulation.

A. Global TSV Assignment

Fig. 6 shows the formulation for the global TSV assignment. In the figure, T_i is the node for the i-th TSV to be assigned to whitespace and W_j is the node for the j-th whitespace block. Since we should assign all TSVs to whitespace blocks, the total amount of flow outgoing from the source equals the number of TSVs and the maximum flow capacity of each edge from the source to T_i is 1. Since edge $s \rightarrow T_i$ has no physical meaning, we set the cost of the edge to zero. Similarly, edge $W_j \rightarrow t$ has zero cost. However, the maximum flow capacity from W_j to the sink equals the number of available TSV slots in whitespace block W_j. The maximum flow capacity from T_i to W_j is 1, which denotes that a TSV is assigned to only one whitespace block. The cost of the edge $T_i \rightarrow W_j$ is computed by the Manhattan distance from T_i to W_j. We solve this minimum-cost flow problem for each die.

If the total amount of flows from whitespace blocks to the sink is less than the total number of TSVs, the problem becomes infeasible. In this case, we manipulate whitespace blocks and go back to the estimation of TSV locations step as illustrated in Fig. 2.

B. Local TSV Assignment

After we assign TSVs to whitespace blocks (global TSV assignment), we assign TSVs to TSV slots in each whitespace block (local TSV assignment) in a similar way. In this local TSV assignment formulation, however, we replace the whitespace blocks (W_j) in Fig. 6 by available TSV slots (S_j) in each whitespace block and the maximum capacity of edge $S_j \rightarrow t$ by 1. The cost of edge $T_i \rightarrow S_j$ is computed by the Manhattan distance from T_i to S_j. We solve this minimum-cost flow problem for each whitespace block.

The reason that we apply global and local assignment separately is because it dramatically reduces the number of variables. If the number of variables is small, however, we can perform the TSV assignment by taking all TSVs and all TSV slots into one assignment formulation.

VI. Whitespace Manipulation

In our signal TSV planning, we need to manipulate whitespace in two cases. First, if we fail to assign TSVs to whitespace blocks, we should insert more whitespace. Second, even if we successfully assign TSVs to whitespace blocks, we could improve the current floorplan by manipulating whitespace. In this section, we present our whitespace manipulation algorithm. Although many papers use concurrent approaches [10]–[12], we manipulate (insert, expand, or redistribute) whitespace blocks one by one.

As a preparation step, we first extract whitespace, create four variables (*left*, *right*, *bottom*, *top*) for each functional block, and create one variable (*demand*) for each whitespace block. Then, for each TSV location found in Section IV, we compute the Manhattan distance from the TSV to each boundary (*left*, *right*, *bottom*, *top*) of each functional block in the same die and add a demand to the four boundaries of the block. To compute the demand, we use the following function:

$$y = \frac{C_{\text{MAX}} - C_{\text{MIN}}}{D_{\text{MAX}} - D_{\text{MIN}}} \cdot (x - D_{\text{MIN}}) + C_{\text{MIN}} \qquad (2)$$

where y is the demand, C_{MAX} is 1.0, C_{MIN} is 0.01, D_{MAX} is $W_{DIE}/6.0$, D_{MIN} to $W_{DIE}/12.0$ where W_{DIE} is the die width, and x is the distance. We also compute the Manhattan distance from each TSV location to each whitespace block in the same die and add a demand to the *demand* variable of the whitespace block using the same demand function.

If the most demanding spot is a boundary of a functional block, we insert a unit whitespace block, which is pre-determined by a user, to the boundary. If the most demanding spot is a whitespace block, we expand the whitespace block by inserting a unit whitespace block to the whitespace block.

VII. Experimental Results

We implement our algorithms using C/C++ and perform all experiments in a 64-bit Linux server with Intel 2.5GHz CPU. To compare our algorithm with [7], we use MCNC and GSRC benchmarks. We also use four industry circuits for more realistic simulation. Table I shows profiles of all the benchmark circuits. Since our algorithms are used in post-floorplanning steps, we develop an in-house 3D

[5]Notice that *top* is always less than or equal to *bot* after the die span computation.

[6]Since we assume that face-to-back stacking is used, if we connect a block in die1 and another block in die3, we insert TSVs in die1 and die2 only.

TABLE IV

COMPARISON OF SINGLE TSV INSERTION, 3D MST-BASED MULTIPLE TSV INSERTION, AND 3D RST-BASED MULTIPLE TSV INSERTION.

		Single TSV insertion		Multiple TSV insertion (3D MST-based)		Multiple TSV insertion (3D RST-based)	
	degree	HPWL-3D ($\times 10^6$)	# TSVs	HPWL-3D ($\times 10^6$)	# TSVs	HPWL-3D ($\times 10^6$)	# TSVs
n100 # nets: 576	3	0.209 (1.00)	1,043 (1.00)	0.168 (0.81)	1,349 (1.29)	0.156 (0.75)	1,165 (1.12)
	4	0.286 (1.00)	1,335 (1.00)	0.226 (0.79)	2,215 (1.66)	0.208 (0.73)	1,841 (1.38)
	5	0.382 (1.00)	1,415 (1.00)	0.294 (0.77)	2,779 (1.96)	0.258 (0.68)	2,258 (1.60)
	6	0.408 (1.00)	1,525 (1.00)	0.329 (0.81)	3,539 (2.32)	0.293 (0.72)	2,826 (1.85)
	7	0.472 (1.00)	1,544 (1.00)	0.439 (0.92)	4,063 (2.63)	0.356 (0.75)	3,256 (2.11)
	8	0.506 (1.00)	1,633 (1.00)	0.483 (0.95)	4,987 (3.05)	0.385 (0.76)	4,004 (2.45)
Geo. mean		(1.00)	(1.00)	(0.87)	(2.38)	(**0.73**)	(**1.69**)
n200 # nets: 1,585	3	0.685 (1.00)	2,918 (1.00)	0.621 (0.91)	3,906 (1.34)	0.539 (0.79)	3,274 (1.12)
	4	0.964 (1.00)	3,544 (1.00)	0.692 (0.72)	5,800 (1.64)	0.609 (0.63)	4,771 (1.35)
	5	1.225 (1.00)	3,816 (1.00)	0.855 (0.70)	7,538 (1.98)	0.757 (0.62)	5,981 (1.57)
	6	1.385 (1.00)	4,241 (1.00)	0.949 (0.69)	9,825 (2.32)	0.832 (0.60)	7,950 (1.87)
	7	1.544 (1.00)	4,287 (1.00)	1.085 (0.70)	11,237 (2.62)	0.946 (0.61)	8,975 (2.09)
	8	1.790 (1.00)	4,516 (1.00)	1.273 (0.71)	13,742 (3.04)	1.017 (0.57)	11,127 (2.46)
Geo. mean		(1.00)	(1.00)	(0.75)	(2.39)	(**0.63**)	(**1.65**)
n300 # nets: 2,000	3	1.035 (1.00)	3,703 (1.00)	0.993 (0.96)	4,876 (1.32)	0.886 (0.86)	4,111 (1.11)
	4	1.685 (1.00)	4,609 (1.00)	1.234 (0.73)	7,538 (1.64)	1.096 (0.65)	6,202 (1.35)
	5	1.671 (1.00)	4,916 (1.00)	1.172 (0.70)	9,860 (2.01)	1.027 (0.61)	7,844 (1.60)
	6	1.933 (1.00)	5,231 (1.00)	1.381 (0.71)	12,203 (2.33)	1.188 (0.61)	9,745 (1.86)
	7	2.105 (1.00)	5,430 (1.00)	1.635 (0.78)	14,449 (2.66)	1.437 (0.68)	11,536 (2.12)
	8	2.362 (1.00)	5,543 (1.00)	2.132 (0.90)	16,865 (3.04)	1.633 (0.69)	13,394 (2.42)
Geo. mean		(1.00)	(1.00)	(0.79)	(2.38)	(**0.68**)	(**1.68**)

TABLE I

BENCHMARK CIRCUITS. # GATES IS THE TOTAL NUMBER OF GATES IN THE BLOCKS, AND # NETS IS THE TOTAL NUMBER OF BLOCK-LEVEL NETS.

	Circuit	# gates	# blocks	# nets	Avg. net degree
MCNC	ami33	-	33	123	4.23
	ami49	-	49	408	2.34
GSRC	n100	-	100	885	2.12
	n200	-	200	1585	2.27
	n300	-	300	1893	2.31
industrial circuits	C1	$75K$	51	6200	2.00
	C2	$92K$	98	1325	4.01
	C3	$278K$	46	1355	2.32
	C4	$566K$	47	2508	2.29

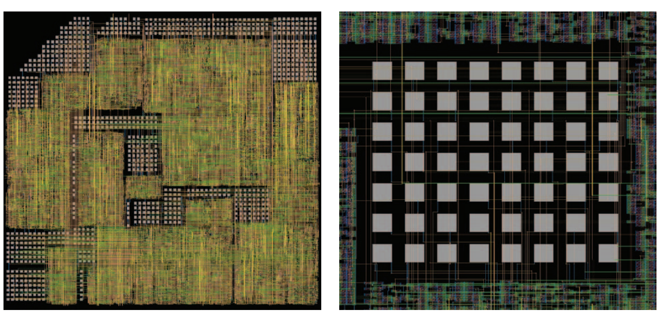

Fig. 7. Full die (top-die) and zoom-in shot of four-die block-level 3D floorplanning (Cadence Virtuoso)

TABLE II

COMPARISON OF 2D AND 3D FLOORPLANNING ON INDUSTRIAL CIRCUITS. THE WIRELENGTH UNIT IS METER. NUMBERS IN PARENTHESES SHOW RATIOS BETWEEN 3D AND 2D WIRELENGTHS. THE TSV DIAMETER IS $2.5\mu m$, THE TSV PITCH IS $4.0\mu m$, AND THE TSV LENGTH IS $20.0\mu m$.

Circuit	2D	# dies	3D		
	HPWL		HPWL-3DBB	HPWL-3D	# TSVs
C1	1.515 (1.00)	2	1.042 (0.69)	1.621 (1.07)	3,080
		3	0.990 (0.65)	1.408 (0.93)	3,976
		4	0.834 (0.55)	1.595 (1.05)	5,864
		5	0.744 (0.49)	1.630 (1.08)	6,169
Geo. mean	(1.00)		(0.59)	(1.03)	
C2	0.375 (1.00)	2	0.274 (0.73)	0.366 (0.98)	1,492
		3	0.221 (0.59)	0.359 (0.96)	2,463
		4	0.198 (0.53)	0.422 (1.13)	3,837
		5	0.174 (0.47)	0.484 (1.29)	4,446
Geo. mean	(1.00)		(0.57)	(1.08)	
C3	0.819 (1.00)	2	0.522 (0.64)	1.380 (0.68)	778
		3	0.369 (0.45)	0.557 (0.68)	1,261
		4	0.404 (0.49)	0.536 (0.65)	1,337
		5	0.332 (0.40)	0.647 (0.79)	2,518
Geo. mean	(1.00)		(0.49)	(0.70)	
C4	2.094 (1.00)	2	1.423 (0.68)	1.479 (0.71)	1,226
		3	1.294 (0.62)	1.496 (0.71)	1,585
		4	1.161 (0.55)	1.491 (0.71)	2,529
		5	0.917 (0.44)	1.320 (0.63)	3,255
Geo. mean	(1.00)		(0.56)	(0.69)	

floorplanner using simulated annealing and 2D sequence pair with inter-die move as well as intra-die perturbation[7] to generate 3D floorplans. Fig. 7 shows a snapshot of the topmost die of a C2 design implemented in four dies.

A. 2D Floorplanning vs 3D Floorplanning

Since all existing works on the comparison of 2D and 3D floorplans use HPWL-3DBB to esimate 3D wirelength, they do not fairly compare 2D and 3D floorplans because HPWL-3DBB significantly underestimates 3D wirelength. In addition, some of them even do not take locations of signal TSVs into account. In this experiment, therefore, we compare HPWL of 2D floorplans and HPWL-3D of 3D floorplans post-processed by our signal TSV planner. To generate 2D floorplans, we run our floorplanner in a 2D mode. To the best of our knowledge, this is the first work on the comparison of 2D and 3D floorplans using the most accurate 3D wirelength metric.

Table II shows that the wirelength (HPWL-3D) of 3D floorplans is slightly longer than that of 2D floorplans by 3% to 8% for relatively small circuits such as C1 and C2. However, the wirelength of 3D floorplans is much shorter than that of 2D floorplans by

[7]Each die has its own sequence pair.

TABLE III
COMPARISON OF SIGNAL TSV PLANNERS. WE REPORT RATIOS BETWEEN OUR RESULTS AND [7] (OURS/ [7]).

# dies	Circuit	WL	# TSVs	# dies	Circuit	WL	# TSVs
3	ami33	0.91	1.26	4	ami33	0.91	1.96
	ami49	0.78	1.26		ami49	0.70	1.34
	n100	0.93	1.03		n100	0.91	1.06
	n200	0.62	0.80		n200	0.82	1.14
	n300	0.75	0.80		n300	0.66	0.82
Geo. mean		**0.79**	**1.01**	Geo. mean		**0.79**	**1.21**

approximately 30% on average for relatively big circuits such as C3 and C4. The reason that 3D floorplans could have longer wirelength than 2D floorplans is twofold. If there are many 3D nets in a 3D floorplan, we need to insert many TSVs, which could significantly increase the die area. The increased die area leads to longer inter-block connections. In addition, if inter-block connections in 2D designs are short, designing this circuit in 3D does not result in shorter inter-block connections.

One thing to notice is that HPWL-3DBB significantly underestimates 3D wirelength. In Table II, HPWL-3DBB is 18% to 47% shorter than HPWL-3D on average. Therefore, we should use HPWL-3D as a wirelength metric for the 3D IC design.

B. Comparison of Signal TSV Planners

Table III shows comparison of wirelength and the number of TSVs between our signal TSV planner and [7]. Since the authors of [7] use HPWL-3DBB, we use HPWL-3DBB as the wirelength metric for fair comparison. We also use the same TSV size as [7] uses. The TSV diameter for MCNC circuits is $20\mu m$ and that for GSRC circuits is $3\mu m$. Since [7] performs signal TSV insertion on fixed-outline floorplans, we run our 3D floorplanning under same constraints – fixed-outline floorplanning with the same whitespace area. We also take I/O pin locations into the wirelength computation.

As Table III shows, our signal TSV planner outperforms [7] by 21% with respect to wirelength for both three-die and four-die floorplans. In addition, the difference between the wirelength of ours and that of [7] increases as the circuit size goes up. For example, we outperform [7] by 9% for ami33. However, for ami49, which is much bigger than ami33, the wirelength of our algorithm is 22% to 30% shorter than that of [7]. We find a similar trend for GSRC circuits. For n100, the wirelength of ours is 7% to 9% shorter than that of [7], but for n200 or n300, we outperform [7] by 18% to 38%. Therefore, we find that our signal TSV planner optimizes wirelength more effectively than [7] as the circuit size goes up.

Since we use multiple TSV insertion, however, we use more TSVs than [7] does. As Table III shows, we use 26% to 96% more TSVs for relatively small circuits such as ami33. However, for large circuits such as n200 and n300, we use slightly more TSVs, or even less TSVs. Since 3D floorplanning has a great effect on the number of TSVs used by signal TSV planners, this result also shows that our 3D floorplanner outperforms the 3D floorplanner used in [7].

C. Single TSV Insertion vs. Multiple TSV Insertion

As mentioned in Section II, multiple TSV insertion can reduce wirelength further than single TSV insertion. In this experiment, therefore, we compare single TSV insertion, 3D minimum spanning tree (MST)-based multiple TSV insertion, and 3D RST-based multiple TSV insertion. For the single TSV insertion, we implement a single TSV insertion algorithm similar to [7]. For the multiple TSV insertion, since the 3D MST is frequently used to find TSV locations [2], we also implement a multiple TSV insertion algorithm using the 3D MST. In this algorithm, we create a 3D MST for each 3D net, and convert each 3D edge into TSV(s), similarly as shown in [2]. In addition, since multiple TSV insertion improves total wirelength effectively for high-degree nets, we generate benchmarks having n nets of degree d. In Table IV, for example, n100 with average net degree 5 denotes that it has 576 nets, and each net is of degree 5.

Table IV shows wirelength and the number of TSVs of these three signal TSV insertion algorithms. As the table shows, 3D MST-based multiple TSV insertion leads to 13% to 25% shorter wirelength on average than the single TSV insertion. In addition, 3D RST-based multiple TSV insertion produces 27% to 37% shorter wirelength on average than the single TSV insertion.

However, since multiple TSV insertion inserts more TSVs than single TSV insertion, the 3D MST-based multiple TSV insertion inserts 2.38× more TSVs on average than the single TSV insertion. Similarly, the 3D RST-based multiple TSV insertion inserts 1.67× more TSVs on average than the single TSV insertion. However, the 3D RST-based multiple TSV insertion uses much less number of TSVs (30% on average) than the 3D MST-based multiple TSV insertion. Therefore, using 3D RST to find optimal TSV locations results in less TSVs and shorter wirelength than using 3D MST.

We also observe in Table IV that wirelength reduction increases as the average net degree goes up. If all nets are two-pin nets (degree 2), no difference exists between single TSV insertion and multiple TSV insertion. However, if all nets are high-degree multi-pin nets (e.g., degree 5), using multiple TSVs helps reduce the total wirelength.

VIII. CONCLUSIONS

In this paper, we proposed a signal TSV planning method to insert signal TSVs effectively. 3D floorplans post-processed by our signal TSV planner show 7% to 38% shorter wirelength than those generated by the state-of-the-art 3D floorplanner. In addition, our 3D RST-based multiple TSV insertion reduces total wirelength more effectively than the single TSV insertion by up to 37%.

REFERENCES

[1] H.-H. S. Lee and K. Chakrabarty, "Test Challenges for 3D Integrated Circuits," in *IEEE Design & Test of Computers*, Sept. 2009, pp. 26–35.
[2] D. H. Kim, K. Athikulwongse, and S. K. Lim, "A Study of Through-Silicon-Via Impact on the 3D Stacked IC Layout," in *Proc. IEEE Int. Conf. on Computer-Aided Design*, 2009.
[3] T. Thorolfsson, G. Luo, J. Cong, and P. D. Franzon, "Logic-on-Logic 3D Integration and Placement," in *Proc. IEEE Int. Conf. on 3D System Integration*, 2010.
[4] M. B. H. *et al.*, "Design and Analysis of 3D-MAPS: A Many-core 3D Processor with Stacked Memory," in *Proc. IEEE Custom Integrated Circuits Conf.*, Oct. 2010.
[5] M. Pathak, Y.-J. Lee, T. Moon, and S. K. Lim, "Through Silicon Via Management during 3D Physical Design: When to Add and How Many?" in *Proc. IEEE Int. Conf. on Computer-Aided Design*, 2010.
[6] X. He, S. Dong, Y. Ma, and X. Hong, "Simultaneous Buffer and Interlayer Via Planning for 3D Floorplanning," in *Proc. Int. Symp. on Quality Electronic Design*, 2009.
[7] M.-C. Tsai, T.-C. Wang, and T. Hwang, "Through-Silicon Via Planning in 3-D Floorplanning," in *IEEE Trans. on VLSI Systems*, 2010.
[8] J. Knechtel, I. L. Markov, and J. Lienig, "Assembling 2D Blocks into 3D Chips," in *Proc. Int. Symp. on Physical Design*, 2011, pp. 81–88.
[9] C. Chu and Y.-C. Wong, "FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design," in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 1, Jan. 2008, pp. 70–83.
[10] X. Tang, R. Tian, and M. D. F. Wong, "Optimal Redistribution of White Space for Wire Length Minimization," in *Proc. Asia and South Pacific Design Automation Conf.*, Jan. 2005, pp. 412–417.
[11] E. Wong and S. K. Lim, "Whitespace Redistribution For Thermal Via Insertion In 3D Stacked ICs," in *Proc. IEEE Int. Conf. on Computer Design*, Oct. 2007, pp. 267–272.
[12] X. Li, Y. Ma, X. Hong, S. Dong, and J. Cong, "LP Based White Space Redistribution for Thermal Via Planning and Performance Optimization in 3D ICs," in *Proc. Asia and South Pacific Design Automation Conf.*, Jan. 2008, pp. 209–212.