Structural Planning of 3D-IC Interconnects by Block Alignment

Johann Knechtel*, Evangeline F. Y. Young**, and Jens Lienig*

* Institute of Electromechanical and Electronic Design, Dresden University of Technology, Germany
** Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong

johann.knechtel@ifte.de, fyyoung@cse.cuhk.edu.hk, jens@ieee.org

Abstract—Three-dimensional integrated circuits rely on optimized interconnect structures for blocks which are spread among one or multiple dies. We demonstrate how 2D and 3D block alignment can be efficiently utilized for structural planning of different interconnects. To realize this, we extend the corner block list and provide effective techniques for 3D layout generation, i.e., block placement and alignment. Our techniques are made available in an open-source, simulated-annealing-based tool. Besides block alignment, it accounts for key objectives in 3D design like fast thermal management and fixed-outline floorplanning. Experimental results on GSRC and IBM-IBM+ circuits demonstrate the capabilities of our tool for both planning 3D-IC interconnects by block alignment and for 3D floorplanning in general.

I. INTRODUCTION

Three-dimensional (3D) stacking of active dies is recognized as a promising approach to meet demands on today's and future chips regarding their performance, functionality, and power consumption. Vertical plugs connecting through separate dies, mainly the through-silicon vias (TSVs), facilitate short and low-power interconnects and thus enable high-performance 3D integrated circuits (3D ICs). 3D network on chip (NoC) architectures have been proposed to increase communication capabilities for logic integration [1] or memory integration [2]. Complementing such approaches, the well-known concept of bus planning, i.e., grouping multiple signals into adjacent wires, remains also relevant for 3D-IC design.

Although the concept of block alignment has been successfully applied in 2D layout representations for bus planning [3, 4], it has been very often neglected in 3D representations. Some studies, e.g., [5–7], enable fixed alignment, i.e., blocks are to be aligned (possibly across several dies) such that their relative positions fulfill fixed distances. However, an application to vertical-bus planning is only indicated in [6]. To the best of our knowledge, none of the existing studies considers alignment ranges, i.e., blocks are to be aligned such that their relative positions fulfill upper and/or lower distance limits. Thus, “flexible” block alignment is not supported so far. We observe that utilizing these different alignment approaches enables structural planning of interconnects for 3D ICs—as illustrated in Figs. 1 and 2, processing block alignment allows one to design dedicated, straight interconnect structures.

To address previously inadequate support for such interconnect structures during 3D floorplanning, we present a methodology based on orchestrated block placement and alignment. In our study, we consider interconnects for different practical scenarios in 3D ICs, as further motivated in Section II.

Our contributions can be summarized as follows.

1. We propose an extension of the corner block list (CBL) (Section IV). Our extension can (i) encode both fixed alignment and alignment ranges, as well as (ii) handle inter- and intra-die alignment in a unified manner. (See Section II for these terms and related background.)
2. We develop effective techniques for 3D layout generation, i.e., block placement and alignment as well as layout packing (Subsection IV-B).
3. We provide an open-source 3D-floorplanning tool based on our CBL extension and simulated annealing (SA) (Section V). Besides block alignment, our tool considers these key objectives: fixed outlines, fast thermal management, layout packing, and wirelength optimization.

II. BUS AND VIA STRUCTURES IN 3D ICs AND RELATED BLOCK ALIGNMENT

As for the 3D design style, we consider block-level integration of 2D blocks. This style is acknowledged as a reliable and efficient approach, especially for first commercial 3D-IC applications [8, 9]. In such 3D ICs, routing paths for (massively parallel) interconnect structures can be enabled by means of block alignment (Figs. 1, 2). Block alignment in 3D ICs can be generally classified into inter-die alignment, i.e., blocks spread
among several dies are to be aligned, and intra-die alignment, i.e., blocks are aligned within one die. The variety of alignment specifics arise from different scenarios for 3D integration and interconnects, which are reviewed next. Note that we focus on signal interconnects in this work. For optimized planning of other interconnect types refer to, e.g., [10].

Monolithic integration has recently gained more interest due to advances in manufacturing processes; for block-level integration, this technology is advantageous in terms of improved interconnectivity [11]. In the general context of massively-interconnected dies, planning vertical buses which connect particular (split-up) blocks spread on separate dies is critical and should thus be considered from early design phases on. It is important to note that TSV-based integration can also exploit such buses, assuming that blocks can be adapted to include TSVs. For example, consider the two macroblocks in Fig. 2: this arrangement of tightly interconnected (for delay and power consumption optimized) modules relies on vertical buses, which are implemented by groups of TSVs. Accounting for such vertical buses during floorplanning requires capabilities for inter-die alignment. That is, in order to include a large number of vertical interconnects, the related blocks have to exhibit some intersecting regions.

A special case of vertical buses are aligned TSV stacks, i.e., TSVs are grouped and placed such that straight interconnects are passing through multiple dies. TSV stacks are relevant for different applications, e.g., to realize regular 3D NoCs, or to limit power-supply noise and to improve thermal distribution [12, 13]. Consideration of aligned TSV stacks during floorplanning requires inter-die alignment with fixed offsets.

Regular (2D) bus structures connecting blocks within dies are independent of the 3D-integration technology. These buses are traditionally considered for several scenarios, e.g., to optimize datapath interconnects. Note that such structures rely on intra-die alignment of related blocks. Depending on fixed / flexible block pins, the planning of 2D buses requires support for fixed alignment / alignment ranges.

III. BASIC PRINCIPLES OF CORNER BLOCK LIST

The corner block list (CBL) [15] is a topological 2D layout representation. In our work, we utilize it mainly for its efficiency (layout generation has a $O(n)$ complexity) and feasible expandability towards a 3D representation (Section IV).

The CBL encodes a floorplan solution as tuple $(S, L, T)$ where $S$ is the block-insertion sequence, $L$ the insertion-direction sequence, and $T$ the sequence of covered T-junctions (see next paragraph). Note that conceptual rooms, i.e., dimensionless entities, are encoded in $S$. Each block is associated with a room; to obtain the physical layout, a transformation from the room topology to block coordinates is required.

During sequential layout generation, two criteria are to be considered for each block (within a room) $s_j \in S$: first, the insertion direction where $l_i = 0$ encodes vertical placement and $l_i = 1$ horizontal placement, respectively; second, the number $t_j$ of T-junctions to be covered. The notion of T-junctions is a verbatim encoding; for example, $t_j = 1$ requires to (perpendicularly) cover the common boundary of two adjacent blocks.

IV. CORBLIVAR: CORNER BLOCK LIST FOR VARIED ALIGNMENT REQUESTS

To enable interconnect structures during 3D IC, we propose an extension of the classical 2D CBL. Our extension is named corner block list for varied alignment requests (Corblivar). It encodes a 3D-IC design integrated on $n$ dies using an ordered sequence $\{CBL_1, \ldots, CBL_n\}$ of CBL tuples and one global alignment sequence $A$. (Thus, Corblivar is a so-called 2.5D layout representation. Refer to [16] for an investigation of several previous 2.5D and 3D representations.)

The alignment tuples $a_k \in A = \{a_1, \ldots, a_n\}$ are designed to encode different types of alignment requests as defined below and illustrated in Fig. 1. Like any layout representation, we need to embed Corblivar in a floorplanning tool; core parts and main features are outlined in Fig. 3.

A. Alignment Tuples

Definition of alignment tuples – Assume the placement of block $s_j$ has to consider some alignment request with regard to (w.r.t.) $s_i$. The request is then defined as tuple $a_k = (s_i, s_j, (AR_x, ART_x), (AR_y, ART_y))$ where $(AR_x, ART_x)$ and $(AR_y, ART_y)$ denote the partial requests with respect to the $x$- and $y$-coordinate. These requests can be independently defined as fixed offset ($ART = 0$), as minimal overlap ($ART = 1$), as maximal distance ($ART = 2$) or as don’t care ($ART = -1$); the meaning of these types is explained next.

Alignment types – Given a fixed offset, $s_j$ is to be placed $AR_x/AR_y$ units to the right/top ($AR_x/AR_y > 0$) or to the left/bottom ($AR_x/AR_y < 0$) of $s_i$, respectively, w.r.t. the blocks’ lower-left corners. Fixed-offset alignment is required for restricted placement, e.g., of blocks with fixed pins.

For a (positive) minimal overlap, the projected intersection of blocks $s_i$ and $s_j$ must be at least $AR_x$ wide and/or $AR_y$ units high. The intention of such alignment is to ensure straight but locally flexible paths for subsequent bus routing / placement of vertical interconnects.

An alignment request defining a maximal distance requires that the center points of blocks $s_i$ and $s_j$ be at most $AR_x/AR_y$ units apart. This way, interconnects structures can be easily limited in their length and/or width.

It may not be necessary to define a request for both $x$- and $y$-coordinates; we label the unrestricted coordinate’s request simply as don’t care.

Dynamic interpretation of requests – Note that the introduced tuples can be easily utilized for both intra- or inter-die alignment by assigning related blocks to one common or to separate CBLS (dies). In other words, the proposed encoding does not restrict blocks to particular dies. For requests spanning multiple blocks (discussed next), it is also possible to combine intra- and inter-die alignment for several blocks.

Definition of tuples to align multiple blocks – For 3D-IC interconnects, implementing links among multiple blocks
is an essential scenario. Thus, let us assume the placement of blocks $s_1 \ldots s_n$ has to consider several, combined alignment requests for interconnects planning. The required set of tuples can be derived in any desired fashion. For example, for requests requiring one reference block $s_1$ (e.g., to represent one specific end of a bus), the tuples would be defined as $(s_1, s_2, (AR_{x_2}, ART_{x_2}), (AR_{y_2}, ART_{y_2})), \ldots, (s_1, s_n, (AR_{x_n}, ART_{x_n}), (AR_{y_n}, ART_{y_n})).$

To give another example, we can encode alignments in a chain-like fashion to enable flexible interconnect structures (i.e., allowing local deviations from a straight, global path): $(s_1, s_2, (AR_{x_2}, ART_{x_2}), (AR_{y_2}, ART_{y_2})), (s_2, s_3, (AR_{x_3}, ART_{x_3}), (AR_{y_3}, ART_{y_3})), \ldots, (s_{n-1}, s_n, (AR_{x_n}, ART_{x_n}), (AR_{y_n}, ART_{y_n})).$

B. Layout Generation

We extend the CBL technique [15] in order to (i) handle inter- and intra-die alignment simultaneously, (ii) consider fixed offsets as well as alignment ranges, and (iii) perform effective layout packing. In the following subsections, we first discuss the orchestration of block placement and alignment and then provide techniques for these steps themselves.

B.1 Orchestration of Block Placement and Alignment

We next discuss the overall process of 3D layout generation. As illustrated in Fig. 3, this requires to (i) manage the layout-generation progress on all dies, (ii) handle the alignment requests, and (iii) interact with block placement and alignment (Subsections B.2 and B.3). In the following, we label "calls" to latter techniques as PLACE and ALIGN, respectively.

Auxiliary data structures – We memorize alignment requests in progress using the alignment stack AS. Progress pointers $p_i = s_i$ denote the currently processed block $s_j$ for each die $d_i$. A die pointer $p = d_i$ is used to keep track of the currently processed die.

Process flow (Algorithm 1) – We perform the following steps for each block $s_i$. Initially, we check whether the associated die $d_i$ is currently marked as stalled (line 5), i.e., layout generation is halted due to another alignment request in progress—this occurs for intersecting requests, i.e., related blocks are arranged in the CBL sequences such that their placement is interfering. To resolve this, we need to unlock die $d_i$—we PLACE the current block $s_i$, mark related changes, and proceed with the next block (lines 6–9). Otherwise (for non-stalled dies), we check if some alignment requests $a_k$ are applying to $s_i$ (line 11). If no $a_k$ are found, we directly PLACE $s_i$ and proceed with the next block (lines 28–29). If some request(s) $a_k$ are defined, we need to handle them appropriately (lines 12–23), as described next. For any given $a_k$, we search the stack AS for it and continue accordingly. Case a: if $a_k$ is found, it was previously handled while processing $s_j$, that is the block to be aligned with $s_i$. Thus, it is assured that preceding blocks on both related dies are placed at this point. We can now safely ALIGN both $s_i$ and $s_j$, mark them as placed, and drop the request $a_k$ (lines 14–17). Note that only in cases where all requests for $s_i$ are handled, we proceed on the current die $d_i$ (line 25). Otherwise, we continue layout generation without loss of generality (w.l.o.g.) on $s_i$‘s die $d_i$ (line 21). Case b: if $a_k$ is not found in AS, $s_i$ was not processed yet. We then memorize $a_k$ as in progress, halt layout generation on $d_i$, and continue on $d_i$’s next die (lines 19–21). Finally, if layout generation is done on $d_i$, we proceed on yet unfinished dies until the whole 3D layout is generated (lines 32–38).

Be aware that deadlock situations, i.e., layout generation on different dies is waiting for each other until particular blocks can be aligned, cannot occur due to resolving of stalled dies. This is true for any alignment request; see also Subsection B.3 for implications on block alignment.

B.2 Block Placement

To maintain a valid layout during placement, it is necessary to consider previously placed blocks. We propose a technique which allows us to (i) efficiently keep track of relevant blocks, (ii) fix CBL tuples w.r.t. exceeding T-junctions, and (iii) virtu-
ally adapt CBL tuples for implicit layout compaction. Our approach differs from [15] in these features but follows the same principle of sequential block placement into dissected rooms.

**Auxiliary data structures** – We keep track of placed blocks using two stacks $H_j / V_j$ for each $CBL_j$. More precisely, these stacks are governed to contain $CBL_j$’s blocks currently covering the vertical right / horizontal upper front of die $d_j$; these are considered as the boundary fronts for further placement.

**Placement flow** – We determine each block’s $s_i \in S_j$ lower-left coordinates $(x_i, y_i)$ as follows. (See Fig. 4 for an example.) First, we retrieve $t_i + 1$ previously placed blocks from the respective stacks $H_j / V_j$. These blocks are referred to as relevant blocks in the following. Note that in cases where only $t_{max} < t_i + 1$ blocks are available, the related CBL tuple is technically infeasible [15]. To fix such invalid tuples, we simply consider all $t_{max}$ blocks in order to fulfill the desired covering of T-junctions as best as possible. Second, we determine $s_i$’s $y / x$-coordinates (orthogonal to the horizontal / vertical insertion direction) by considering the structural change of $CBL_j$’s room dissection. In case a new column / row is implicitly defined due to covering all relevant blocks during placement of $s_i$, we set the respective $y / x$-coordinate to 0. In the remaining cases, we derive the coordinate from the relevant blocks’ lower / left front. This can also be thought of as placing a new column / row into the existing room dissection. Third, we determine $s_i$’s $x / y$-coordinates (along the insertion direction) by considering the right / upper front of previously placed blocks which are intersecting with $s_i$ in its orthogonal, recently determined $y / x$-coordinates. Fourth, we update the placement stacks to follow the changed layout’s fronts as follows. We push $s_i$ onto the insertion-direction-related stack $H_j / V_j$. In case $s_i$ is not covered by some relevant block to its top / right front, we also push $s_i$ to the (unrelated) stack $V_j / H_j$. Finally, we push relevant blocks not covered by $s_i$ back to $H_j / V_j$; these blocks remain part of the layout’s boundary front and are thus to be furthermore considered.

**Virtual CBL adaption** – For any block smaller than the room it is supposed to cover, the next, adjacent block(s) will be packed “into the room” of this smaller block (Fig. 5). We refer to this feature as virtual CBL adaption since it results in practice in different CBL tuples encoding the same (compact) layout. Note that virtual CBL adaption is generally applied during block placement.

### B.3 Block Alignment: Inter- and Intra-Die Alignment

Recall that our alignment tuples support different alignment types and can be interpreted as inter- or intra-die requests.

**Scenario I: both blocks are placed** – In this case, we cannot fulfill $a_k$ since we omit post-placement shifting.

**Scenario II: one block is yet unplaced** – Here, we assume w.l.o.g. that $s_i \in a_k$ is yet unplaced and $s_j \in a_k$ was previously placed. Depending on both the coordinates of (placed) $s_j$ and the properties of $a_k$, we may be able to fulfill $a_k$ as follows. First, we determine $s_i$’s $y / x$-coordinates orthogonal to its horizontal / vertical insertion direction (“Second”, Subsection B.2). Next, based on both the inherent offset between $s_i$ and $s_j$ and the defined alignment of $a_k$, we derive the required shifting range $rs(s_i, s_j)$ / $rs(s_j, s_i)$, i.e., the remaining offset of $s_i$ w.r.t. $s_j$ in order to fulfill $a_k$. Note that $rs(s_i, s_j) = -rs(s_j, s_i)$, i.e., the shifting range is directed and invertible. In cases $rs(s_i, s_j) = 0$, $a_k$ is already fulfilled. In cases $rs(s_i, s_j) < 0$, we would need to shift $s_i$ downward / leftward which is trivially prohibited while maintaining a valid layout. Alternatively, we could shift placed $s_j$ upward / rightward;

---

1We would like to stress that this limitation only applies to particular configurations. That means, adapting the CBL configurations during alignment-aware 3D floorplanning (as proposed in Section V) can resolve this issue.

2Based on our observations, shifting placed blocks most likely requires adjacent (or nearby) blocks to be shifted as well in order to maintain a valid layout. This is impractical in the presence of different alignment requests—such shifting can then undermine handling of remaining requests and even invalidate previously processed ones.
Required Shifting Ranges:

**Fulfilled:**
\[ rs_{x,y}(s_7, s_4) = 0 \]

**Failed:**
\[ rs_{x,y}(s_8, s_5) = - rs_{x,y}(s_5, s_8) = - 30 \]

V. 3D-FLOORPLANNING TOOL

We provide Corblivar along with our C++ implementation of a SA-based 3D-floorplanning tool [18]. Our holistic concept of (orchestrated) block placement and alignment differs notably from previous works. Applying SA-based floorplanning during initial experiments, we observe limitations of existing techniques w.r.t. solution-space exploration for block alignment as well as for “classical” 3D floorplanning. Thus, some effective extensions are needed; notable features of our tool are (i) a SA framework including two optimization phases and specific cost models, (ii) an adaptive SA cooling schedule, and (iii) a fast yet sufficiently accurate thermal analysis.

A. Optimization Criteria & Cost Models

We next discuss applied optimization criteria along with their cost functions/models. Note that cost functions are formulated for SA’s classical cost-minimization approach, i.e., lower cost correspond to more optimized layouts.

**Outline** – This criteria unifies evaluation of the layout’s bounding boxes, i.e., packing density, as well as fixed-outline fitting. This is achieved by extending Chen and Chang’s aspect-ratio-based cost model [19]. Our model is defined as

\[
\begin{align*}
C_{OL} &= c_{PD} + c_{AVR} \\
C_{PD} &= \frac{1}{n} \alpha \left( 1 - \frac{n_{\text{feasible}}}{n} \right) \times \max_d \left( \frac{\text{OL}}{\text{OL}(b \notin d_i)} \right) \\
C_{AVR} &= \frac{1}{n} \beta \left( 1 - \frac{n_{\text{feasible}}}{n} \right) \times \max_d \left( \Delta_{\text{AR}}(d_i) \right) \\
\Delta_{\text{AR}}(d_i) &= AR_{\text{outline}}(b \in d_i) - AR_{\text{outline}}(d_i)
\end{align*}
\]

where \( c_{PD} \) and \( c_{AVR} \) denote the respective cost terms for packing density and aspect-ratio violation. Functions \( A_{\text{outline}} \) and \( AR_{\text{outline}} \) determine the outline’s area and aspect ratio w.r.t. a set of blocks \( b \in d_i \) / a die \( d_i \), respectively. Note that we perform cost calculation for previous \( n \) layout operations where \( n_{\text{feasible}} \leq n \) operations resulted in a valid layout, i.e., blocks on all dies are fitting into the fixed outline.

**Wirelength and TSV count** – We assume that the lowermost die \( d_0 \) is connected to the package board. For each net \( n \), we determine the half-perimeter wirelength (HPWL) on each related die \( d_i \) separately, denoted as \( HPWL(n, d_i) \). To do so, we construct the bounding box by encircling connected terminal pins (only for \( d_0 \)) and assigned blocks on \( d_i \) and on the upper die \( d_j, j > i \) as well. The latter is required to model wires for connecting blocks with TSV landing pads in upper dies.

**Overall cost terms are defined as**

\[
\begin{align*}
C_{WL} &= \sum_n \left( l_{TSV} \times TSV_s(n) + \sum_{d_i \in n} HPWL(n, d_i) \right) \\
C_{TSVs} &= \sum_n TSV_s(n)
\end{align*}
\]

where \( TSV_s(n) \) denotes the required TSV count for net \( n \). Note that we also account for “TSV wirelength” \( l_{TSV} \) in \( C_{WL} \).

**Thermal management** – Cost \( c_T \) is the estimated maximal temperature of the critical die furthest away from the heatsink. Details on thermal modelling are given in Subsection E.

**Alignment mismatch** – For an alignment tuple \( a_k \), we describe the spatial mismatch between desired alignment and actual layout as cost \( c_{AMM}(a_k) = |rs(s_i, s_j)| \) (Subsection IV-B.3). Overall cost is then calculated as \( c_{AMM} = \sum_{a_k \in A} c_{AMM}(a_k) \).

B. Optimization Phases

We consider two different phases for SA optimization; these phases support efficient solution-space exploration and layout optimization for 3D floorplanning with block alignment.

**Phase I, “Fixed-Outline Fitting”** – The cost function is defined as \( c_{FOF} = c_{OL} \). Note that we do not perform alignment (Subsection IV-B.3) in this phase. The reason for initially focusing SA’s search solely on the fixed-outline is simply that non-fitting layouts are a “knock-out”, regardless of any achieved block alignment and layout optimization. The transition to phase II is made when the SA search triggers the first, fixed-outline fitting layout.

**Phase II, “Alignment and Layout Optimization”** – We compose the cost function as

\[
\begin{align*}
C_{ALO} &= c_{OL} + (1 - \alpha) \times \max_{c' \in C'} \left( \beta c_{WL}/c_{WL_{init}} + \gamma c_{TSVs}/c_{TSVs_{init}} \right) \\
\delta(c_T/c_{T_{init}}) \times \epsilon(c_{CAMM}/c_{CAMM_{init}})
\end{align*}
\]

with \( \beta + \gamma + \delta + \epsilon \leq 1 \). Note that we memorize initial cost terms like \( c_{WL_{init}} \) during transition to phase II, i.e., we derive them from the first valid solution. Furthermore, note that we consider \( C_{OL} \) as essential term in this phase as well; based on our experiments, the SA search for comprehensively optimized layouts still depends on outline fitting/optimization.

C. Layout Operations

We consider the following set of layout operations to support the SA heuristic in effective exploration of Corblivar’s solution space: swapping blocks within or across dies (CBL sequences), swapping or moving whole CBL tuples within or across CBL sequences, switching a block’s insertion direction, switching a block’s T-junctions, rotating hard blocks, and guided shaping of soft blocks as proposed in [19].
For optimization phase I, operations and blocks / CBL tuples are selected randomly. In phase II, blocks related with failed alignment requests are particularly selected. These blocks are swapped with adjacent blocks such that \(|rs(s_i, s_j)| \) is reduced, i.e., such that the alignment is more likely to be fulfilled.

D. Cooling Schedule

As indicated earlier, we require an adaptive cooling schedule for improved efficiency of solution-space exploration. Our schedule is capable of (i) guiding the SA search within the global phases and (ii) escaping local minima. The schedule is composed of three different phases, explained below. Note that \(i \) labels the current step of \(i_{\text{max}} \) total temperature steps.

Phase “Adaptive Cooling” – We apply this cooling phase during SA phase I, which is aiming for fixed-outline fitting.
\[
T_{i+1} = \left( c_{f1} + \frac{i-1}{i_{\text{max}}-i_{\text{first}}} \right) \times T_i
\]
The cooling rate slows down (given that \(c_{f1} < c_{f2} < 1.0 \)); our intention here is to achieve initially fast cooling for the global scope, followed by slower cooling in a confined, “local” solution space.

Phase “Reheating and Freezing” – This is applied for SA phase II, i.e., after a fitting layout was found in step \(i_{\text{first}} \).
\[
T_{i+1} = \left( 1 - \frac{i-1}{i_{\text{max}}-i_{\text{first}}} \right) \times c_{f3} \times T_i
\]
The cooling rate increases steadily; however, setting \(c_{f3} > 1.0 \) results in an initial reheating for \(i \geq i_{\text{first}} \). This way, the SA search has an increased flexibility for accepting high-cost solutions in this “interesting solution-space region” covering the first fitting layout. According to experiments, this limits the risk for being subsequently trapped in solution-space minima.

Phase “Brief Reheating” – This phase enables a somewhat “automatized” and robust cooling schedule.
\[
T_{i+1} = c_{f4} \times T_i, \quad c_{f4} > 1.0
\]
It is applied in alternation with the phase “reheating and freezing” for individual temperature steps during SA phase II. Such brief reheating helps the SA search to escape local minima; it is applied when we observe \(\sigma(C_{\text{LAGO}}) \approx 0 \) during previous \(k \) steps, that is when the search reached a local “cost plateau”. This technique is inspired by Chen and Chang’s study [19]; their approach, however, proposes reheating solely at one particular temperature step, which we believe is not as effective as our cost-controlled reheating.

E. Fast Evaluation of Thermal Distribution

For fast yet accurate (steady-state) temperature analysis, we extend the work of Park et al. on power blurring [20]. Instead of using computationally intensive finite-differences or finite-elements analysis (FEA), power blurring is based on simple matrix convolution of thermal impulse responses and power-density distributions. (Park et al. reveal promising results when comparing to ANSYS FEA runs; they achieve maximal errors of less than 2% with computation speedups of \( \sim 60 \times \).)

For improved efficiency and to provide an integrated floorplanning tool [18], we refrain from time-consuming FEA runs for retrieving thermal masks [20]. Instead, we model the masks’ underlying thermal impulse responses as 2D gaussian functions \(g(x, y, w, s) = w \exp \left(-\frac{x^2}{2w^2}\right) \exp \left(-\frac{y^2}{2s^2}\right) \) with \(w \) as amplitude-scaling factor and \(s \) as lateral-spread factor. To obtain the whole set of required masks [20], we need some scaling measure for \(g \). We thus adapt \(w \) for each die \(d_i \)’s mask such that \(w_i = w_i / (w_0) \). where \(w_0 \) denotes a scaling parameter. For actual parametrization of \(w_0, w_s \) and \(s \), we determine for each different 3D-IC setup (w.r.t. die count and dimensions) (i) an exemplary thermal distribution using a 3D-IC extension of HotSpot [21], a state-of-the-art academic thermal analyzer, and (ii) a best fit for above parameters based on a local search using HotSpot’s solution as reference model.

VI. EXPERIMENTAL RESULTS

We conducted several experiments described below to validate Corblivar’s capabilities. Relevant configuration details are given in Subsection A; results are discussed in Subsection B.

Structural planning of interconnects – We consider a set of several interconnects running both within and across dies; the (arbitrarily defined) set contains 10 width- and length-limited buses, each covering up to 5 blocks, along with 3 block pairs to be vertically aligned. We assume that each interconnect structure bundles 64 signals. For structural planning of these interconnects, in total 18 blocks have to be aligned simultaneously; related alignment tuples can be retrieved from [18]. Such scenario has not been considered in previous studies, thus we cannot meaningfully compare to other work.3

Regular and large-scale 3D floorplanning – To evaluate Corblivar’s efficiency w.r.t. key 3D floorplanning objectives, we look into layout packing, wirelength and thermal optimization. (Note that we refrain from deriving alignment tuples for the considered benchmarks’ nets. In other words, here we do not apply block alignment for interconnects planning and/or wirelength optimization.) We compare our work to relevant previous studies [22, 23]. Furthermore, we demonstrate Corblivar’s scalability by utilizing the IBM-HB+ benchmark suite [24]. To the best of our knowledge, this is the first time that these large-scale circuits are considered for 3D floorplanning.

A. Configuration

3Previous studies on block alignment for 3D ICs have looked into differing scenarios. Nain and Chrzanoska-Jeske [5] propose techniques to split up and align (sub)modules among adjacent dies with fixed (zero) offsets. They neglect to provide derived benchmarks containing split-up blocks, thus a comparison is hindered. Law et al. [6] consider a more flexible problem formulation; for vertical bus planning, they define sets of blocks for each die separately and require (at least) one block from each set to be vertically aligned with one block from the other sets. This simplified alignment problem is not compatible with our approach where we require all specified blocks to be aligned. Li et al. [7] indicate capabilities for block alignment but refrain from providing further details and related experimental results. Finally note that all aforementioned studies exclusively consider vertical alignment with fixed offsets.
B. Results

Structural planning of interconnects – We observe that the entire set of interconnects is successfully integrated, i.e., all related blocks can be simultaneously aligned (upper part of Table I). Compared to experiments where planning of interconnects is ignored (lower part of Table I), we expect and observe an increase of die outlines and deadspace—block alignment limits the flexibility of layout packing. More importantly, however, we observe notable wirelength increases in case of neglected interconnects planning; these overheads arise from routing detours for interconnects embedded in unaligned blocks. Finally, fixed die outlines were fulfilled in any case, i.e., the proposed SA optimization phases are effective.

An example for successful interconnects planning with corresponding block alignment is illustrated in Fig. 7.

Regular 3D floorplanning – Next, we discuss results on conducting floorplanning with applied layout packing and (equal) consideration of thermal and wirelength optimization (Table III). We observe that Corblivar is competitive with a force-directed tool [22] and superior to a SA-based tool [23]: both represent state-of-the-art academic works. In particular, we achieve comparable wirelengths and temperatures as [22] but with reduced die outlines and deadspace ratios. This indicates the efficiency of layout packing, which is most likely achieved by the proposed virtual CBL adaption. Comparing to [23], however, we note that Corblivar’s layouts exhibit larger deadspace ratios and thus reduced packing densities. Nonetheless, we achieve reduced wirelengths in most cases. Also, the high packing density of [23] comes at a price; maximal temperatures are notably increased by tens of Kelvins compared to Corblivar. Thus, our tool effectively addresses the trade-off between packing density and maximal temperature. Furthermore, fixed outlines were fulfilled in these experiments as well.

As for our temperature analysis, we observe that it shows some local deviations compared to HotSpot-verification runs (Fig. 8). As indicated in [20], convolution-based thermal analysis particularly induces estimation errors at die boundaries. Thanks to our proposed mask parametrization, the actual thermal-distribution scale (i.e., the scale w.r.t. HotSpot runs)

TABLE I

<table>
<thead>
<tr>
<th>Metric</th>
<th>2 Dies</th>
<th>3 Dies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n100</td>
<td>n200</td>
</tr>
<tr>
<td></td>
<td>n100</td>
<td>n200</td>
</tr>
<tr>
<td>Wirelength (cm x 10^3)</td>
<td>1.18</td>
<td>1.81</td>
</tr>
<tr>
<td>Total Deadspace (%)</td>
<td>29.21</td>
<td>30.39</td>
</tr>
<tr>
<td>Wirelength (cm x 10^3)</td>
<td>1.00</td>
<td>1.08</td>
</tr>
<tr>
<td>Total Deadspace (%)</td>
<td>18.82</td>
<td>27.04</td>
</tr>
<tr>
<td>Runtime (s)</td>
<td>80</td>
<td>359</td>
</tr>
</tbody>
</table>

TABLE II

<table>
<thead>
<tr>
<th>Metric</th>
<th>2 Dies</th>
<th>3 Dies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sb1</td>
<td>sb3</td>
</tr>
<tr>
<td></td>
<td>sb1</td>
<td>sb3</td>
</tr>
<tr>
<td>Wirelength (cm x 10^3)</td>
<td>4.77</td>
<td>7.29</td>
</tr>
<tr>
<td>Total Deadspace (%)</td>
<td>17.24</td>
<td>19.26</td>
</tr>
<tr>
<td>Runtime (s)*</td>
<td>1195</td>
<td>3611</td>
</tr>
</tbody>
</table>

matched nevertheless. For analysis during layout optimization, i.e., maximal-temperature estimation, our approach is thus applicable. It is also efficient due to fast computation; one run can be conducted in ~20ms. (For comparison, one HotSpot run can take tens of seconds up to a few minutes.)

Large-scale 3D floorplanning – The IBM-HB+ suite does not include power information; we thus configured Corblivar only for wirelength and packing optimization (including successful consideration of fixed-outline constraints). Results on arbitrarily selected circuits are provided in Table II. We observe that total deadspace for these experiments is on average larger than for experiments on some GSRC circuits. This is expected and likely due to the fact that IBM-HB+ circuits contain up to ~1,500 blocks where largest blocks are ~33,000 times bigger than smallest ones; such designs are difficult to floorplan [26].

VII. Summary

In this work, we extend 3D floorplanning towards structural planning of interconnects—an important yet inadequately addressed scenario (future) massively interconnected 3D ICs. To tackle this omission of previous works, we promote block alignment. We initially discuss how 3D (inter-die) and 2D (intra-die) alignment can be applied for planning of diverse interconnects like vertical buses connecting (split-up) blocks on separate dies or classical 2D buses. We then introduce Corb-
### Table III: Comparative Results on GSRc Benchmarks for Layout Packing with Thermal and Wirelength Optimization – Benchmarks are Not Enlarged for Fair Comparison

<table>
<thead>
<tr>
<th>Metric</th>
<th>Corblivar, 2 Dies</th>
<th>Corblivar, 2 Dies</th>
<th>Corblivar, 3 Dies</th>
<th>Corblivar, 3 Dies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n100</td>
<td>n200</td>
<td>n500</td>
<td>Avg</td>
</tr>
<tr>
<td>Wirelength (µm × 10³)</td>
<td>3.70</td>
<td>6.57</td>
<td>9.07</td>
<td>6.45</td>
</tr>
<tr>
<td>Die Outlines (µm² × 10³)</td>
<td>0.10</td>
<td>0.99</td>
<td>1.61</td>
<td>1.20</td>
</tr>
<tr>
<td>Total Deadspace (%)</td>
<td>11.98</td>
<td>12.01</td>
<td>15.65</td>
<td>13.21</td>
</tr>
<tr>
<td>Max Temp [°(K)]</td>
<td>313.81</td>
<td>314.53</td>
<td>315.95</td>
<td>314.76</td>
</tr>
<tr>
<td>Runtime (s)</td>
<td>108</td>
<td>286</td>
<td>548</td>
<td>314</td>
</tr>
</tbody>
</table>

Liviar, a 3D layout representation based on an extended corner block list with novel alignment tuples. To this end, we also develop effective techniques for block placement and alignment. We note that it is essential to synchronize alignment across the whole 3D IC—in particular, inter-die alignment requires to consider each die’s layout in progress. Our techniques handle this appropriately for different scenarios of blocks to be aligned and/or to be placed. We embed Corblivar into an open-source, SA-based floorplanning tool; we also develop necessitated extensions like adaptive SA cooling and convolution-based fast thermal analysis. Experimental results on GSRc and large-scale IBM-HB+ benchmarks demonstrate Corblivar’s applicability for structural planning of interconnects, i.e., block alignment, as well as its competitive performance for “classical” 3D floorplanning while considering fixed outlines, layout packing, thermal and wirelength optimization.

### Acknowledgments
The work of J. Knechtel was supported by the German Research Foundation (Project 14011/1). For the parametrization of the thermal model (Subsection V-E), Timm Amstein provided Octave scripts which are included in [18]. The authors thank Igor L. Markov for comments on drafts of this paper.

### References