Multiple Dice Working as One: CAD Flows and Routing Architectures for Silicon Interposer FPGAs

Ehsan Nasiri, Javeed Shaikh, Andre Hahn Pereira, and Vaughn Betz, *Member, IEEE*

***Abstract*—Multiple FPGA dice can be connected through a silicon interposer to form a larger system than one monolithic die can accommodate. Such systems are now commercially available but many open questions remain concerning their key architecture parameters and efficiency, as the signal count between dice is reduced and the signal delay between dice is increased compared to a monolithic FPGA. We create a new version of VPR to target interposer-based FPGAs and investigate modifications to placement and routing and the incorporation of partitioning into the flow to improve results. Our best CAD flow reduces the routing demand for interposer FPGAs with realistic amounts of connectivity between dice by 47% and improves the circuit speed by 13% on average. Architecture modifications to add routing flexibility when crossing the interposer are very beneficial and improve routability by a further 11%. With these CAD and architecture enhancements, we find that if an interposer supplies (between dice) 40% of the routing capacity that the normal (within-die) FPGA routing channels supply, there is negligible impact on circuit routability. Smaller interposer routing capacities do impact routability however: minimum channel width increases by 10% and 70% when an interposer supplies only 20% and 10% of the within-die routing, respectively. The interposer also impacts delay, increasing circuit delay by 11% on average for a 1 ns interposer signal delay and a two-die system.**

***Index Terms*—Algorithms, FPGA, Silicon interposer, 2.5D ICs.**

I. Introduction

Interposer-based FPGA systems are composed of multiple FPGA dice connected through a silicon interposer. The interposer uses an older fabrication technology than the FPGA dice, and links between dice include both a micro-bump on each FPGA die and a metal wire on the interposer [1]. This results in reduced connectivity and increased delay between dice compared to the routing bandwidth and delay across a cutline within a single die. Such interposer systems are sometimes called 2.5D devices, since they make use of vertical stacking of dice on an interposer to enable higher integration levels [2]. In this work we present an architecture study of silicon interposer-based FPGAs that analyzes how the reduced connectivity and increased delay impact their performance.

Silicon interposers enable the creation of large FPGAs from smaller dice, which can both improve yield and fabricate FPGAs that are too large to be manufacturable on a single die. Making large FPGAs from multiple smaller dice is particularly interesting at the beginning of a new manufacturing process, when defect densities are high. In this case, good-die yield drops dramatically as the die size increases, drastically impacting the availability of large FPGAs early in the process lifetime.

The idea of using 3 dimensions to design FPGAs is not new. [3] proposed the creation of 3D FPGAs by stacking 2D FPGAs and connecting them with solder bumps. Lin et al also studied 3D FPGAs, but using a different approach [4] that utilized multiple active layers with different FPGA functions (logic, routing, and configuration) in each. While promising, both these approaches have manufacturing challenges: [3] requires a higher density of through-silicon vias than can currently be manufactured and the multiple active layers required by [4] are not yet widely available. In contrast, silicon interposers linked to dice via microbumps are now manufacturable.

Chaware et al presented Xilinx's approach to silicon interposer FPGAs [5]. They describe the physical characteristics of their implementation, including the bump pitch and estimates of the amount of die-to-die connectivity and the die-to-die delay. However [5] does not analyze the architecture question of the routability of the resulting system, nor describe possible CAD optimizations, which are the questions we investigate.

The main contributions of our work are as follows:

- VPR CAD tool modifications to model and optimize for 2.5D silicon interposer-based FPGAs.
- An investigation of the efficacy of incorporating partitioning into various stages of the CAD flow.
- A study of how multi-FPGA system routability is impacted by the amount of between-die connectivity and by the switching flexibility at the FPGA die - interposer boundary.
- An analysis of the interposer's impact on timing.

The paper is organized as follows. Section II gives more information about silicon interposer technology. Section III describes the changes made to the FPGA architecture description and VPR to target 2.5D FPGAs. Section IV describes optimizations to the VPR placement and routing algorithms for interposer FPGAs and Section V describes how partitioning can further enhance the CAD flow. Section VI presents architectural results for 2.5D FPGAs.

An earlier and less detailed version of this work appeared in [6]. We greatly extend this earlier work by enhancing the CAD tools, incorporating partitioning into the flows and investigating a broader range of architecture questions.

E. Nasiri is with Altera, Toronto, ON, Canada (email: enasiri@altera.com)
J. Shaikh is with Google, Mountain View, CA, USA (email: jvds@google.com)
A. H. Pereira is with the Computer and Digital Systems Engineering Dept., University of São Paulo, SP, Brazil (email: andre.hahn@usp.br)
V. Betz is with the Dept. of Electrical and Computer Engineering, University of Toronto, ON, Canada (email: vaughn@eecg.utoronto.ca)
Manuscript received January 6, 2015.

II. Silicon Interposer Background

Interposer-based FPGAs allow systems larger than a single die, making a "More than Moore" improvement on the size and number of logic cells possible, and with chips combined with far more connectivity and less delay than if they were in separate packages connected through a printed circuit board.

The improvement in the capacity of 2.5D FPGAs over conventional ones is very significant. Xilinx's largest 28 nm interposer-based FPGA contains 4 FPGA dice and 1.954 million logic cells [7], while the largest non-interposer Virtex-7 die has 979k logic cells and Altera's largest 28 nm FPGA has 952k logic elements [8]. Even though all these FPGAs use a 28 nm process, silicon interposer technology allows the creation of FPGAs with twice the resources possible on even an extremely large single die. Xilinx's recently announced Virtex UltraScale line of 20 nm FPGAs makes even more aggressive use of interposers; the largest monolithic FPGA in this family contains 941k logic cells, while the four largest capacity FPGAs are all built with silicon interposers and span a range from 1.25 to 4.43 million logic cells [9].

Another major advantage of interposer-based FPGAs comes at the beginning of a new manufacturing process, when the defect density is high [1]. To illustrate this impact consider a new process in which the defect density is $1/cm^2$ (a reasonable value early in the process lifecycle) and the die area is $6cm^2$, which roughly matches the size of the largest member of a high-end FPGA family such as Virtex 7. Using the Poisson Yield Model [10], the yield is only 0.25% of dice. If instead the chip is composed of four $1.5cm^2$ dice, the yield is 22%. This means that a 12 inch silicon wafer with $730cm^2$ of area would produce on average 0.3 working $6cm^2$ dice, while the same wafer would produce on average 107 working $1.5cm^2$ dice. Therefore, as a $6cm^2$ chip would be composed of four $1.5cm^2$ dice, the wafer would yield 26.75 systems on average, as the "assembly yield" of placing these four die on an interposer is very high [5]. Hence the number of interposer-based FPGAs created from the same silicon wafer would be almost $100\times$ greater than a monolithic FPGA of the same size, greatly impacting not only cost but also the availability of such a large FPGA.

When the process matures and the defect density decreases this yield advantage drops. Consider a mature process with defect density of $0.1/cm^2$. The yield for a $6cm^2$ die is 55% and the yield for a $1.5cm^2$ die is 86%. Hence the number of single die FPGAs created from a $730cm^2$ silicon wafer would be 66.9 and the number of interposer-based FPGAs created would be 104.6. While the interposer-based FPGA still has a yield advantage it might not lead to a major cost advantage when the cost of the interposer and assembling the die to it are included. For the large, state-of-the-art FPGAs that are built early in a process cycle and heavily used for prototyping, however, there is clearly a compelling cost advantage to an interposer-based solution.

A. Virtex Interposer-based FPGAs

The Xilinx 2.5D FPGAs from the Virtex-7 and Virtex Ultrascale families are currently the only commercially available silicon interposer-based FPGAs [11]. As described in Section III we are studying the impact of several key interposer parameters on the performance of the multi-die system, including (i) the percentage of the wiring normally present between rows of the FPGA which is cut when crossing between dice, and (ii) the extra delay (vs. a normal connection between adjacent rows) added when one must traverse the interposer. To locate where Virtex-7 lies in this architecture space, we combined published information on the implementation of Xilinx's interposer-based FPGAs [5] with a detailed analysis of the *XC7V2000T* FPGA routing resources visible in the *Vivado Device View*.

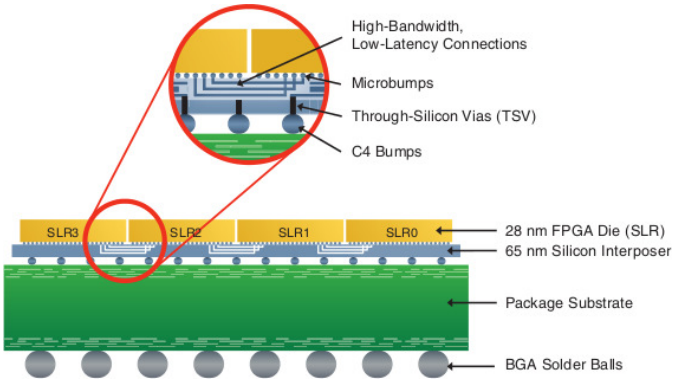


Fig. 1. Lateral view of an interposer-based FPGA [11]. The FPGA dice are at the top, and are connected to the silicon interposer through microbumps. The interposer is then connected to the substrate through C4 bumps.

The *XC7V2000T* is composed of four identical dice arranged such that the vertical routing crosses between the dice. Each horizontal edge of each die has 280 groups of 48 length-12 wires crossing the interposer, which sums to a total of 13440 wires between dice. There are also 40 clock wires crossing the interposer. The average vertical routing channel in this FPGA contains 210 wires and there are approximately 280 vertical channels, resulting in approximately 58800 vertical wires crossing a horizontal cutline within a die. Hence the number of wires which cross the interposer is about 23% of the total number of within-die vertical wires.

The 28 nm dice are connected to the 65 nm silicon interposer through microbumps with a 45μm pitch. Hence the area occupied by microbumps at one edge of one die is $13440 \times (45\mu\text{m})^2 = 27\text{mm}^2$. If we assume each die is $7 \times 12mm$, as presented by Chaware et al. in [5], the bumps have to be spread out near the edge and need to go as far as $2.25mm$ away from the edge of the die. This greater distance from the border increase the length of the inter-die connections, and along with the presence of the micro-bumps and their capacitance, leads to an increased delay for these crossing wires vs. that of a typical on-die routing wire. Chaware et al. state that the latency to cross the interposer is approximately $1ns$. For comparison, a typical medium length 28 nm FPGA routing wire (e.g. spanning four logic blocks) has a delay of approximately 125 ps, while a longer wire (e.g. spanning 12 logic blocks) has a delay of approximately 250 ps.

Overall, these interposer FPGAs have increased delay and reduced connectivity between dice, with approximately 23%

of the usual number of vertical wires between dice and approximately $1ns$ of increased delay to cross the interposer.

III. Architecture Models

To properly model a silicon interposer FPGA, we use version 7.0 of the popular FPGA exploration toolset, Verilog-to-Routing (VTR) [12]. The logic synthesis portions of the flow (ODIN II and ABC) are used as-is, while we modify the placement and routing portion of the flow (VPR [13]) to model and optimize for interposer-based FPGAs. The modifications are made in such a way that they require no changes to any of the input files, so one can experiment with interposer-based FPGAs with any existing benchmark circuits and any existing VPR-format FPGA architecture description simply by specifying appropriate command-line parameters.

We add the following parameters to VPR: *number of cuts*, *% wires cut*, *increased delay*, and *interposer-routing interface*. These parameters describe the interposer portion of the 2.5D FPGA as detailed below.

A. Number of Cuts

Number of cuts describes how many cuts are made to the interposer-based FPGA versus a monolithic die. If *number of cuts* equals 1 then there are 2 dice, and so on. We investigate values of 1 and 3 (2 and 4 dice). The Virtex 7 family has members with *number of cuts* $= 2$ and 3. Figure 2 shows a sample architecture with one cutline.

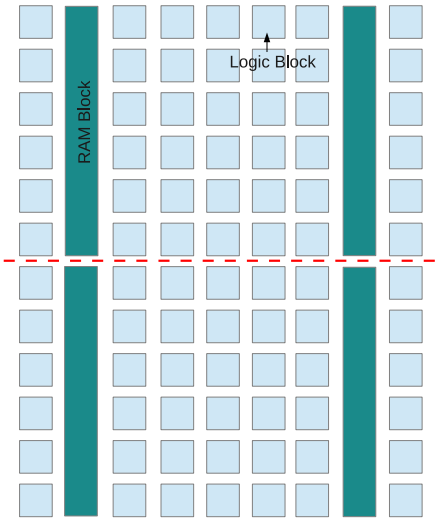


Fig. 2. A sample two-die / 1-cutline architecture containing both logic blocks and RAM blocks.

B. % Wires Cut

This variable describes and models the reduced connectivity between different dice by specifying the fraction of routing wires that are removed at the border between dice. For example, if a channel had 200 wires and *% wires cut* was 70, 140 of them would be cut and only 60 would pass through the interposer. Higher values of *% wires cut* make an interposer easier to manufacture, and can reduce the interposer delay by allowing all the microbumps linking dice to be placed near the die edge. However, the higher *% wires cut* is, the less routable the multi-die system becomes. As described in II-A, the Virtex 7 family has *% wires cut* $= 77\%$.

C. Increased Delay

Interposer wires are longer and wider than on-die wires and have microbumps on each end. *Increased delay* models the resulting larger delay when compared with wires which are internal to a die. A reasonable estimate for this variable is around $1ns$, as presented by Chaware et al [5].

D. Interposer-Routing Interface

In a simple architecture, the vertical routing wires at the edge of an FPGA die that can not connect to an interposer wire are dangling. Since interposer wires are scarce, we want to be able to study architectures in which the interposer wires are more accessible. To do so, we have enhanced the router to support the following interposer-routing interface options.

1) Fan-in Transfer for Interposer Wires
   For an architecture with *% wires cut* $= 0$, all vertical routing wires have a corresponding interposer wire that connects them to the adjacent die. Practical interposers, however, can only accommodate a fraction of all vertical routing wires. When *fanin transfer* option is *on*, we place a multiplexer at the input of each available interposer wire and connect adjacent "dangling" vertical routing wires to the multiplexer. This enables additional flexibility in routing by ensuring that *every* vertical routing wire could *potentially* drive an interposer wire through a multiplexer. As an example, if *% wires cut* $= 75\%$, 3 of every 4 interposer wires are cut. Therefore, we place a 4-to-1 mux at the input of the one interposer wire that is not cut, and we connect the 3 vertical routing wires that do not have a corresponding interposer wire to the multiplexer.
2) Fan-out Transfer for Interposer Wires
   When *fanout transfer* option is *on*, we allow an interposer wire to drive not only its own corresponding vertical wire at the other side of the cut, but also other "dangling" vertical routing wires that are not driven by anything because their corresponding interposer wires are cut.
3) Bidirectional Interposer Wires
   Dominant routing architectures in modern FPGAs use unidirectional wires [14], [15] which can only be driven at one point, normally their start point. When this option is *on*, we allow interposer wires to be driven from one end or the other by adding tri-state buffers at both ends.

These options are independent and can be combined to achieve more flexible routing. Enabling any of these options requires adding specific circuitry (such as multiplexers) on the dice which causes very small area and delay increase that we neglect in our experiments.

E. Implementation

To model an interposer-based multi-FPGA system in the CAD tool, we use the existing VPR infrastructure for monolithic FPGAs and make several modifications to it.

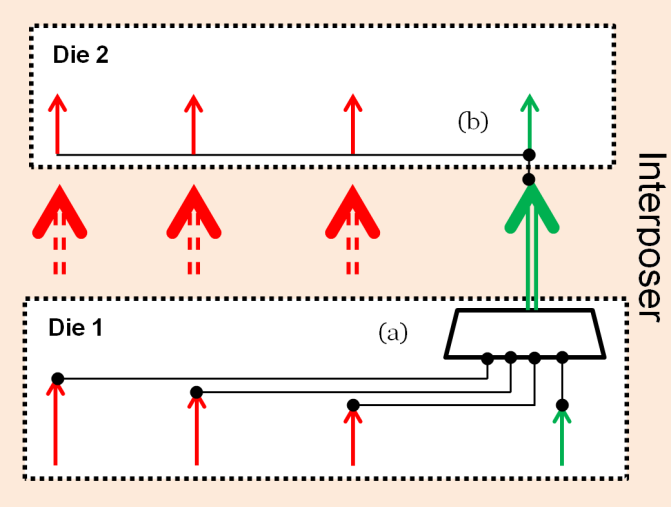


Fig. 3. For a channel width of 4 and 75% wires cut, only 1 interposer wire exists (in green) and 3 interposer wires are cut (in the red). (a) Fan-in Transfer for Interposer Wires allows red routing wires in die 1 to drive a mux that allows them to cross the interposer layer. (b) Fan-out Transfer for Interposer Wires allows the output of the interposer wire to drive wires in other tracks in die 2.

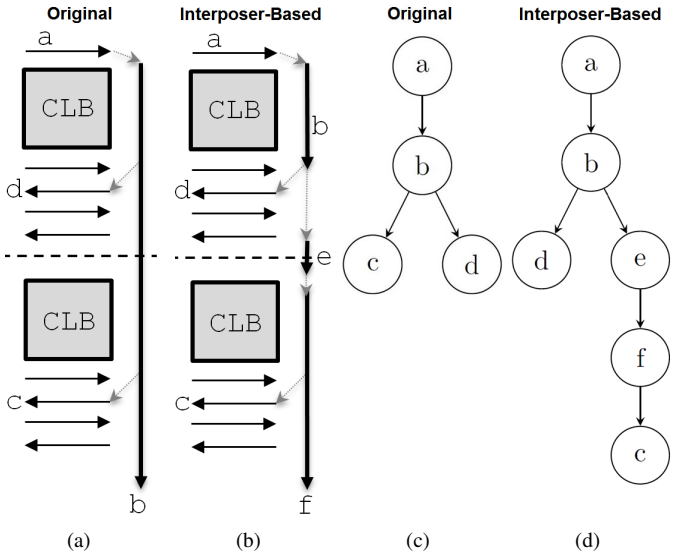


Fig. 4. (a) Regular routing wire connectivity. (b) Replacing the long wire with two wires and connecting them via an interposer wire. (c) rrgraph for regular connectivity. (d) rrgraph after modifications.

First, the presence of multiple dice in an interposer-based FPGA is modeled by creating horizontal cuts in the FPGA. The *number of cuts* is specified by the user, and we use that to determine the y-location of the cuts. The cutlines "snap to the grid" such that the height of all dice are equal and no cutline cuts through a block. For instance, a RAM block of height 4 must remain a unit and can not be sliced by a cutline.

Second, knowing the location of the cutlines, the placer uses the *(x,y)* coordinate of any pin or block to determine the die to which it belongs. The placer uses this information as well as *interposer delay* to better decide where to place each netlist block in order to minimize the cost of inter-die signal transfer as described in Section IV-A.

Third, the Routing Resource Graph (*rrgraph*) must be modified. The rrgraph is the data structure that defines all the available routing wires (*rrnodes*) and switches in the FPGA, as well as the delay of each [16]. Given a suitable rrgraph, the VPR router can implement circuits in the desired FPGA, and the VPR timing analyzer can estimate their delay. In the experiments below, we use an architecture that has only unidirectional wires.

Previous work by Pereira et al [6] modeled the interposer indirectly: they did not introduce new rrnodes in the rrgraph to represent the interposer wires. Rather, they found the rrnodes that cross a cutline. Then they either removed all outgoing edges of these rrnodes (if the routing node was part of the *% wires cut*), or, they increased the switch delay of the rrnode fan-outs (if the wire was not cut) to account for the additional interposer delay.

While this is a reasonable method to model such architectures, it makes it difficult to experiment with different routing-interposer interfaces such as placing input muxes for interposer wires. Therefore, we modify the rrgraph differently to more accurately reflect the interposer layer directly:

For every vertical routing wire in the channel, we add a new routing node ("*interposer node*") into the rrgraph. Then, we determine rrnodes (e.g. vertical routing wires) that cross a cutline in the rrgraph, and we replace them with 2 rrnodes: one that is above the cutline and one that is below the cutline. We properly transfer fan-ins and fan-outs of the original node onto the two newly created nodes and connect the two nodes using an interposer node. Finally, we remove *all* edges from *%wires cut* percent of interposer nodes and increase the switch delay of output edges of the remaining interposer nodes by *increased delay*. Figure 4 shows an example of these operations.

As illustrated in Figure 5, no vertical routing wires cross a cutline boundary and a fraction of them connect to other dice via interposer wires.

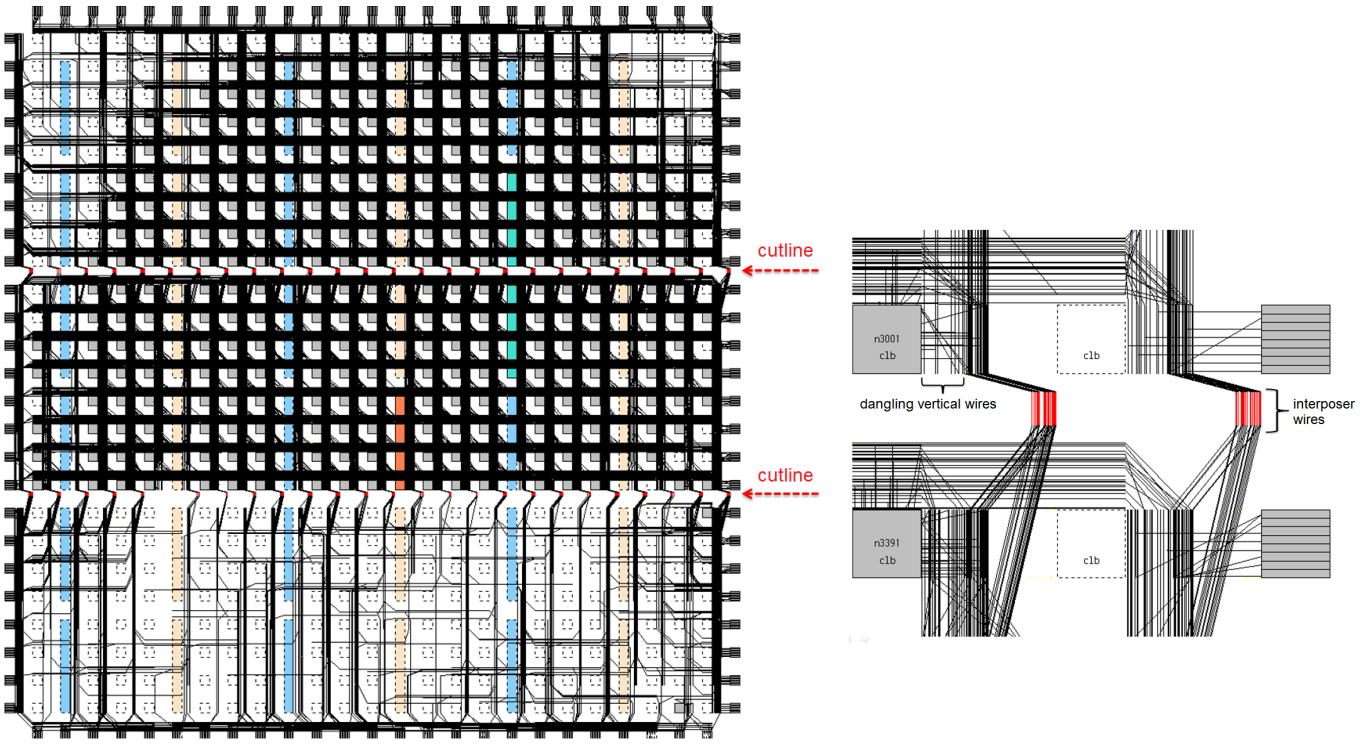
IV. CAD Enhancements

A. Placer Optimization

Placement is crucial in mitigating the impact of the reduced wiring and increased delay when crossing between dice; a good placement should minimize the number of signals crossing between dice, particularly time-critical ones. VPR uses 2 different costs as the metrics for its placer algorithm: the timing cost and the bounding box (wiring) cost. We modified the placer's bounding box and timing costs to account for the increased latency to cross the interposer and for the reduced wire capacity close to the cutlines, as the wire availability becomes more sparse.

1) Placer Timing Cost: The usual VPR timing cost is a (criticality-weighted) summation of the estimated delay (given the current placement) of every connection required by the circuit [17]:

$$Cost_T = \sum_{\forall i,j \subset circuit} delay(\Delta x_{ij}, \Delta y_{ij}) \times Criticality(i,j) \quad (1)$$



(a) (b)

Fig. 5. VPR Graphical User Interface. (a) Full chip view. (b) Connectivity at the cutline. Due to the limited signal capacity of the interposer, some vertical routing wires are left dangling.

where *ij* denotes a connection from block i to block j that exists in the circuit netlist.

This cost function assumes that the FPGA is homogeneous and consequently the delay between 2 points (x_1, y_1) and (x_2, y_2) only depends on $(\Delta x, \Delta y)$. This is obviously not true for interposer-based FPGAs, as the cutlines make them heterogeneous in the y direction. To solve this problem and improve the quality of the results, we add an extra term to the delay function:

$$delay(i,j) = delay(\Delta x_{ij}, \Delta y_{ij}) + N_{crossed}(i,j) \times delay\_increase \quad (2)$$

where $N_{crossed}(i,j)$ is the number of times this path has to cross the interposer to go from (x_i, y_i) to (x_j, y_j) and $delay\_increase$ is the timing penalty of crossing the interposer.

2) Placer Wiring Cost: The bounding box cost estimates the amount of wiring required for a net, based on the number of pins and size of the net's bounding box. VPR's original formulation is [13]:

$$Cost_{W\_orig} = \sum_{n=1}^{N_{nets}} q(n) \times [\frac{bb_x(n)}{W_{chanx}} + \frac{bb_y(n)}{W_{chany}}] \quad (3)$$

where $bb_x(n)$ and $bb_y(n)$ are the dimensions of the bounding box of net *n* in the x and y directions, respectively. W_{chanx} and W_{chany} are the average x-directed and y-directed channel widths over this bounding box. Finally, $q(n)$ is a function obtained from [18] which models the fact that bounding boxes underpredict the required routing for high fanout nets. $q(n)$ slowly increases with the fanout of net *n*.

VPR's wiring cost also considers the FPGA to be homogeneous, and uses only the number of nets, the size of the net's bounding box and the average number of wires per channel to calculate the cost. Thus, to account for the reduced connectivity near the cutlines we create an extra cost term, $Cost_{cut}$. This new cost is added to (3) to create the total wiring cost.

$$Cost_W = Cost_{W\_orig} + Cost_{cut} \quad (4)$$

In [6], Pereira et al tested several different $Cost_{cut}$ formulations, and different weighting for each. They found that the following $Cost_{cut}$ results in the best quality of results in terms of minimum routable channel width and critical path delay.

$$Cost_{cut} = \sum_{n=1}^{N_{nets}} C' \times bbHeight(i) \times N_{crossed}(n) \quad (5)$$

where

$$C' = \frac{ratio\_wires\_cut}{W_{chany}} \quad (6)$$

bbHeight(n) is the height of the bounding box of the net *n* and $ratio\_wires\_cut$ is the ratio of wires cut at the cutline.

This formulation of C' ensures that the new cost term is of roughly the same magnitude as $Cost_{W\_orig}$ in (3) and that it is weighted more heavily for interposer architectures in which the wiring between dice is more scarce.

This cost function performs better than several others as shown in [6], and we believe the key to the good performance of this cost functions is that $Cost_{cut}$ produces gradual gains as bounding boxes crossing the cutline shrink to be closer and closer to being captured entirely on one side of the cutline. Consider for example the 3 bounding boxes shown in Figure 6. Note that bounding box (a) and (b) both cross the cutline, but (b) is penalized less by (5) because bounding box (b) is mostly on the lower side of the cutline; it is more likely that later placement changes will result in the bounding box moving entirely below the cutline, reducing interposer wiring demand. Cost function (5) penalizes bounding box (a) more than (b) and (b) more than (c) to guide placement to gradually move bounding boxes to one side or the other of a cutline.

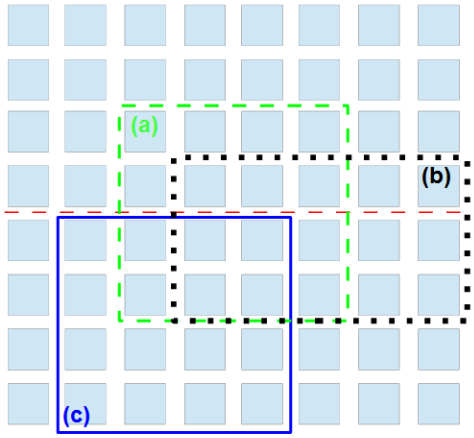


Fig. 6. Illustration of three different scenarios for the wiring cost. The dashed green box shows a case where *(a)* it crosses the interposer, the dotted black box shows a case where *(b)* it barely crosses the cutline, and the solid blue one shows a case where *(c)* the bounding box does not span the cutline.

B. Router Optimization

Once the Routing Resource Graph is modified as detailed in Section III, the VPR router adapted automatically to the interposer architecture. The router starts at a net source node and expands by discovering adjacent routing nodes until it reaches the destination node. At any point, the router knows the cost of the path taken to the current point and it uses a *lookahead cost function* to estimate the resource and delay cost from the current node to the destination. Using this lookahead cost, the router can properly sort routing alternatives. The lookahead cost function in VPR is not aware of the interposer, and therefore it takes a long time to converge to a routing decision. To make the lookahead aware of the interposer, we modify the lookahead cost function by adding terms that account for extra *interposer delay* and extra *interposer nodes* that may exist on the route from the current node to the destination node.

The VPR router uses the following to calculate a cost for a routing node *r* on the route from *i* to *j*:

$$Cost(r) = Criticality(i,j) \times Delay_{Elmore}(r, topology) + [1 - Criticality(i,j)] \times Congestion(r) \quad (7)$$

Given this cost function, the router's lookahead algorithm determines the total cost of the route from *i* to *j* when routing node *r* is used:

$$Cost_{Total}(r) = \sum_{\forall l \in RT(i,r)} Cost(l) + \alpha \times ExpectedCost(r,j) \quad (8)$$

where *RT(i,r)* is the set of all routing nodes used so far from *i* to *r* and α is a constant number that determines the aggressiveness of the directed search. α is 1.2 by default in VPR. VPR's original $ExpectedCost(n,j)$ function is [16]:

$$ExpectedCost(r,j) = Criticality_{fac} \times T_{delay} + (1 - Criticality_{fac}) \times ExpectedCongestionCost(r,j) \quad (9)$$

T_{delay} is an estimate of the delay that the router predicts it will see on the route from *r* to *j*. Since (9) assumes a homogeneous FPGA, T_{delay} does not include the additional delay some routes experience by going through the interposer. Thus, for interposer-based architectures, we replace T_{delay} in (9) with T'_{delay} to include the delay of expected interposer hops from *r* to *j*.

$$T'_{delay} = T_{delay} + T_{interposer} \times ExpectedInterposerHops \quad (10)$$

We modify $ExpectedCongestionCost(r,j)$ in (9) in a similar way to account for the new interposer routing nodes.

The interposer-aware router lookahead makes routing decisions more quickly given the large delay difference of a path that crosses the interposer as opposed to a path that doesn't.

C. Effectiveness of the Enhancements

For all experiments in this paper, the remaining architecture parameters of the FPGA are taken from the "flagship" architecture of the VTR project (*k6_frac_N10_mem32K_40nm.xml*) [12]. The parameters of this architecture are in line with both current commercial FPGAs and academic research into best practices. It consists of logic clusters with 10 fracturable 6-LUTs per block (N=10, k=6), and also includes 32kb RAM blocks and DSP blocks configurable to perform 9x9, 18x18 or 36x36 multipliers. The delay values in the architecture are taken from 40 nm circuit simulations and 40 nm commercial FPGAs. It uses unidirectional routing, with all wire segments having length L = 4.

We used VPR 7.0, with the enhancements we detailed above, and the architecture file *k6_frac_N10_mem32K_40nm*

in the experiments below. All experiments targeted the smallest FPGA (with number of rows equal to number of columns) which could accommodate a benchmark circuit; this represents a very full FPGA with little white space left, and hence presents a difficult case to an interposer-based FPGA as no die can be left mostly empty.

We have used the eight largest circuits from the VTR benchmark [19], namely: *bgm*, *LU8PEEng*, *LU32PEEng*, *mcml*, *mkDelayWorker32B*, *stereovision0*, *stereovision1* and *stereovision2*. The size of these circuits ranges from $9,100$ to $153,000$ primitives (LUTs, FFs, etc.), with an average size of $52,600$ primitives. All results are the geometric mean over all circuits for a given interposer architecture.

To obtain the *critical path delay* the circuits were run with a low stress routing with a channel width, $W = 1.3 \times minW$, where $minW$ is the minimum channel width for which the circuit is still routable.

1) Effectiveness of Placer Optimizations: Enabling the enhancements described in Section IV-A improves both the minimum routable channel width and circuit speed. As shown in Figure 7, for architectures with low *% wires cut* (high interposer capacity), the placer optimizations have only a marginal benefit on routability compared to the original VPR placer since the interposer wires are not scarce, but as more wires are cut and we reach the realistic interposer capacity of commercial FPGAs, the placer optimizations provide higher and higher reductions in minimum channel width. When 80% of wires are cut, the placer optimizations result in 11.3% reduction in minimum channel width.

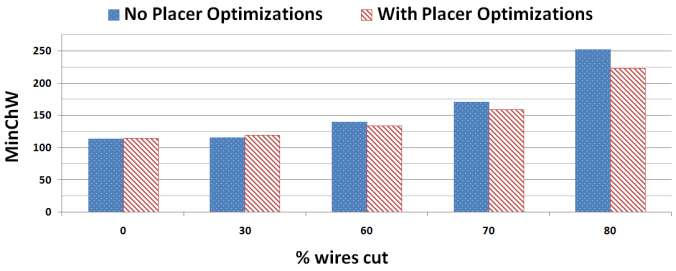


Fig. 7. Impact of placer optimizations on minimum channel width.

The optimizations also guide the placer to keep related entities on the same die to reduce the number of inter-die signal crossings and hence increase circuit speed. This speed improvement is fairly consistent across a wide range of % wires cut, averaging 12.5% higher circuit speed.

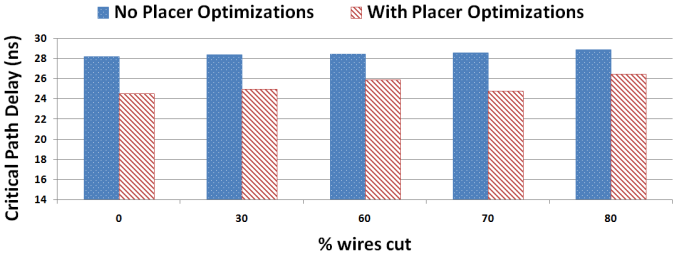


Fig. 8. Impact of placer optimizations on circuit speed.

2) Effectiveness of Router Optimizations: The router optimizations described in Section IV-B greatly reduce the runtime of the router. Figure 9 shows the geomean of the router runtime for the 8 largest VTR circuits for different *% wires cut* values. On average, the router runs 31 times faster and finds identical minimum channel widths for all circuits.

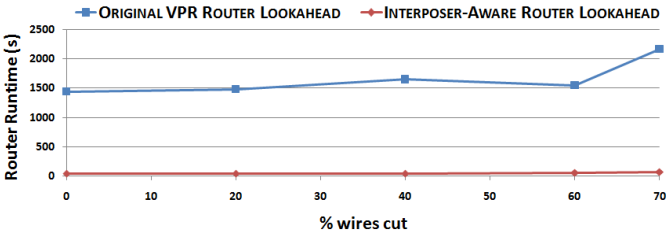


Fig. 9. Router runtime with original and updated lookahead function.

The new router lookahead changes the priority of the routes discovered by the router. Using the new router lookahead function increases the critical path delay by 1.3% on average. We consider this small quality loss acceptable and use the enhanced router lookahead in all our experiments due to its greatly reduced runtime.

V. Adding Partitioning to the CAD Flow

Given that the number of signals that can cross the interposer between FPGA dice is considerably more limited than within an FPGA, using a partitioner to divide the circuit into one partition per die is a promising CAD flow.

A. CAD Flow Variations

The overall CAD flow, including the partitioning steps, is shown in Figure 10.

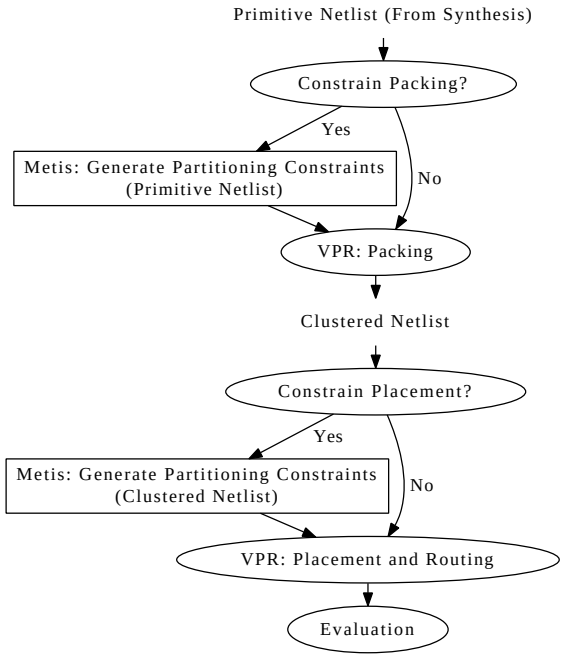


Fig. 10. Possible ways to add partitioning to the VPR CAD flow.

1) Partitioner Stage: As shown in Figure 10, we explore the effect of circuit partitioning at two different stages of the CAD flow. The first possibility is to partition the netlist of primitive elements (LUTs, FFs, RAM slices, etc.) before packing into the larger function blocks such as logic blocks and (32 kb) RAM blocks. The partitioning constraints must be respected by the packer; it must not pack two primitives assigned to different partitions (dice) into the same function block. We updated the VPR clustering algorithm to respect this new constraint. The second possible way to incorporate partitioning is after packing, where partitioning is applied to the function block (cluster-level) netlist. Each partition is then assigned (randomly) to a separate FPGA die. The partitioning constraints must then be respected by the placer; it is free to move function blocks around within a die, but not across interposer cutlines. These two partitioning flow options can be combined – if partitioning is applied before packing, one can also pass the partition information to the placement stage. The post-packing placement constraints can be derived from the primitive block partitioning constraints, with the valid placement region for a given function block being equal to the intersection of the valid placement regions of the primitives packed into that function (clustered) block. There are, therefore, four possible partitioning flows:

1) Leaving both the packer and the placer unconstrained. This is the base flow that does not use partitioning, as described in Section IV.
2) Partitioning the primitive netlist to guide the packer, leaving the placer unconstrained.
3) Leaving the packer unconstrained, and partitioning the function block (clustered) netlist to guide the placer.
4) Partitioning the primitive netlist to guide the packer, and, as described above, passing these constraints to the placer.

2) Partitioning Tool: Two well-known freely available graph partitioning tools are Metis [20] and hMetis [21]. Metis operates on graphs, while hMetis operates on hypergraphs – a hypergraph is a generalization of a graph, where each edge can connect any number of vertices. It is most natural to represent a circuit netlist as a hypergraph, where the vertices are netlist pins (or blocks) and the hyperedges are the nets joining the pins. Owing to the natural hypergraph representation, hMetis would be the most appropriate choice of partitioner. However, there are other issues to consider for our use case.

Graph partitioning algorithms assign vertices to partitions while satisfying "balance constraints". The balance constraints exist to drive the algorithm away from the trivial partitioning solution where all vertices are in the same partition, leaving all other partitions empty. An example of a basic balance constraint is one which requires an equal number of vertices in each partition. Generally such a perfect balance overly restricts the solution, so some fractional "unbalance" is allowed; in this work we use 0.05 (5%). For two partitions, $P1$ and $P2$, this translates to [22]

$$\frac{|P1|}{|P1 \cup P2|} \in [0.5 - unbalance, 0.5 + unbalance] \quad (11)$$

To derive an appropriate balance constraint for our use case, consider that FPGA circuit netlists may contain different block types such as LUTs, registers, RAM slices, and multipliers, among others. The basic balance constraint of (11), applied to a netlist with only LUTs and multipliers, would ensure that the total block count (sum of the LUT count + multiplier count) in each partition is roughly balanced. The problem with (11) in the case of multiple block types is that the capacity, per block type, of each partition is not taken into account. It is therefore possible to generate partitions that cannot be realized because the number of resources of each type is not balanced across partitions, even though the total number of resources is balanced. For example, there is nothing preventing the partitioner from moving all LUTs to partition 1 and all multipliers to partition 2, which may not be realizable given the per-block-type capacity of each partition.

One solution is to enforce a balance constraint on each block type, i:

$$\forall i, \left| \frac{|P1_i|}{|P1_i \cup P2_i|} - 0.5 \right| < unbalance \quad (12)$$

where $P1_i$ is the set of all blocks of type i in partition 1. We refer to this type of per-block set of constraints as heterogeneous balance constraints. Note that there are complex legality constraints governing which LUTs and FFs can be packed into legal logic blocks, as well as which multipliers can be packed into legal DSP blocks [12]. As the partitioner is not aware of these constraints, flow 3 above (which runs the partitioner after function block packing) will be able to more precisely balance resource use across partitions.

The current version of hMetis does not support heterogeneous balance constraints, though Metis does. Metis was used in our CAD flow. However, as the circuit netlist is naturally represented as a hypergraph, it needs to be transformed to a graph before Metis can process it.

3) Hypergraph to Graph Transformation: We transformed the circuit netlist hypergraph to a graph in the following way:

1) For each hyperedge of the circuit netlist hypergraph, we generated a graph (the *per-net subgraph*).
2) We combined all of the per-net subgraphs, summing the edge weights for edges appearing in more than one per-net subgraph, to generate the total netlist graph.

We explored several ways to generate the per-net subgraphs. The two parameters in this operation are the graph topology and the edge weight scheme. We considered two graph topologies, clique and star, as illustrated in Figure 11. We assigned the same edge weight to every edge of the generated subgraph. Edge weights were computed based on the number of vertices in the originating hyperedge, n, and we considered edge weights equal to 1, $1/n$, and $1/n^2$. We use *hyperedge model* to refer to the combination of graph topology and edge weight scheme.

Figure 12 shows how a netlist of three blocks and two nets is transformed into its component subgraphs.

To select the best hyperedge model, we used the Metis partitioner to generate graph partitions (which assign each vertex to a partition number) and applied the partitioning result to the original untransformed hypergraph. To rank the

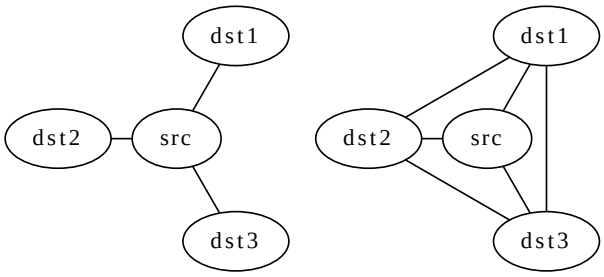


Fig. 11. (a) Star and (b) clique graph topologies.

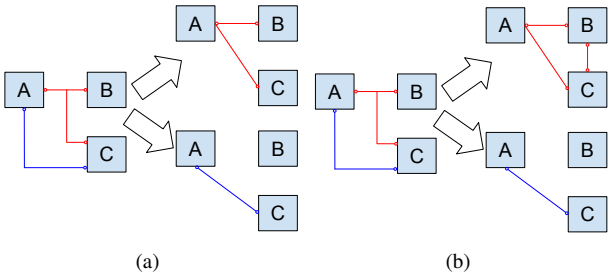


Fig. 12. The hypergraph to graph transformation with (a) the star topology, and (b) the clique topology.

different models, we computed the *hyperedge cutsize* for every hyperedge model, across the same circuits as in Section IV-C. The hyperedge cutsize is the number of hyperedges crossing the cutline, as illustrated in Figure 13. As it equals the number of nets that cross the interposer boundary, it is a measure of circuit routability. The results are shown in Figure 14.

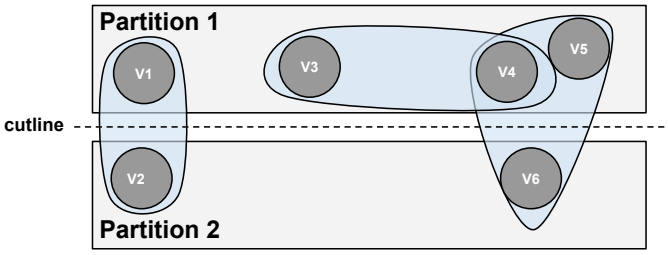


Fig. 13. A graph with six vertices, three hyperedges, and two partitions. As there are two hyperedges crossing the cutline, the hyperedge cutsize is 2.

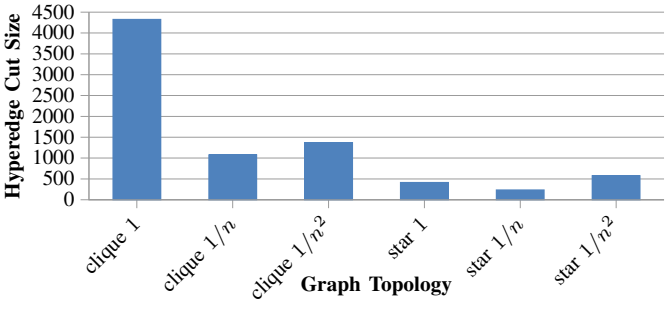


Fig. 14. Hypergraph cut size achieved by Metis with different hyperedge models and weightings. n is the number of vertices on a hyperedge.

The performance of the clique topology is significantly worse than that of the star topology, regardless of edge weight scheme. An intuitive explanation for this result is that with the clique topology, the partitioner does not have knowledge of which vertex is the source of a net and which vertices are sinks. In contrast, the star topology clearly differentiates sources from sinks and this appears to give the partitioner an anchor point that pulls all hyperedge fanouts toward the source. Additionally, if n is the average number of vertices in a hyperedge, the clique topology generates $O(n^2)$ edges while the star topology generates only $O(n)$ edges, so the star topology saves both memory and runtime.

For both topologies, the $1/n$ edge weight scheme gives the smallest cut size on average. This can be explained intuitively in terms of the total edge weight over all edges of the graph. The star 1 (constant) edge weight scheme assigns a total weight of n to each net, which heavily weights high-fanout nets. In contrast, the star $1/n$ model assigns the same total weight of 1 to each net. The star $1/n^2$ model penalizes high-fanout nets relative to lower-fanout nets. Since we seek to minimize the hyperedge cut, it makes intuitive sense to weight all hyperedges equally, and the star $1/n$ model achieves this.

To validate the choice of hyperedge cutsize as a proxy for circuit routability, we ran the partitioning CAD flow for each hyperedge model, across several circuits. We use an unbalance of 5%, split the clustered netlist into two partitions (constraining only the placer), cut 80% of the wires crossing the interposer, and impose a $1ns$ delay penalty for wires crossing the interposer. To accommodate unbalance in the placement engine, we increase the size of the grid of complex blocks that make up the FPGA device. Relative to the minimum device size required for placement (without partitioning constraints), we add 10% to the complex block grid width and 10% to the grid height. Figure 15 shows the geometric mean of the minimum channel width required for a successful route, for each hyperedge model.

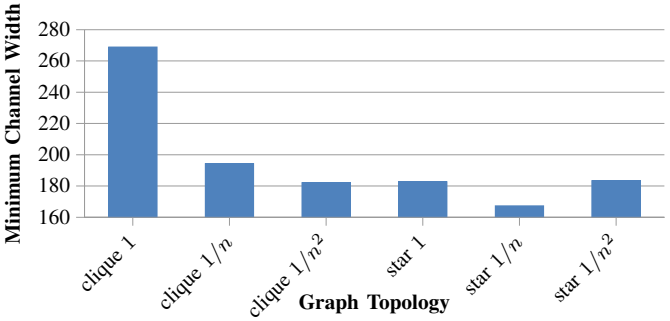


Fig. 15. Minimum channel width achieved by VPR with different hyperedge models and weightings. n is the number of vertices on a hyperedge.

The star topology with 1/n edge weights achieves the best minimum channel width, confirming that hyperedge cutsize is a good proxy for routability. Consequently, we use this model in all future results in this paper. Unlike routability, we found that the post-routing critical path delay was not strongly impacted by the hypergraph model used.

4) Partitioner Stage Results: We compare the performance of the four CAD flow variations described in Section V-A1 on the 8 largest VTR benchmarks. We again use an unbalance

of 5% and split the clustered netlist into two partitions (constraining the packer and/or the placer, as required by the flow under test). The interposer parameters (% wires cut = 80% and delay increase = 1 ns) and bloat factor (10% in each dimension) are the same as in Section V-A3. Figure 16 shows the minimum routable channel width for each partitioning CAD flow variation.

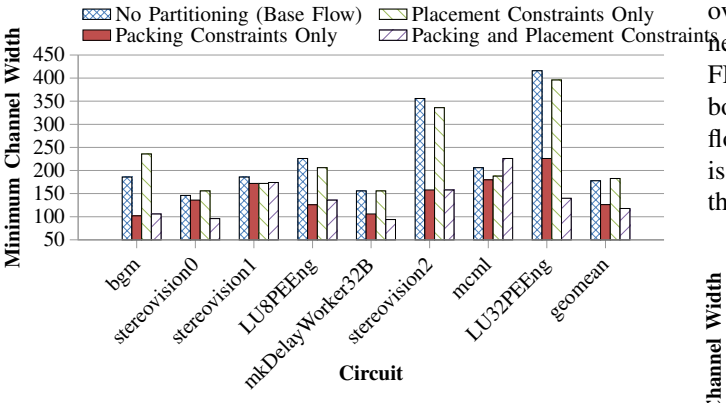


Fig. 16. Minimum channel width achieved by VPR for each partitioning flow.

On average, the best minimum channel width was obtained when constraining both packing and placement. However, constraining packing alone performed significantly better than constraining placement alone. This shows that the VPR packer strongly benefits from partitioning constraints when targeting an interposer-based architecture, likely because it forbids the packing of primitives that are not naturally related into one function block. Interestingly, other recent work targeting conventional FPGAs has shown that using a partitioner to guide packing benefits routability [23]. On the other hand, The VPR placement algorithm with the enhancements described in Section 4 sees only a modest routability benefit from the partitioning constraints.

For the same set of circuits, we computed the critical path delay at a channel width equal to 1.3x the minimum. The results are shown in Figure 17.

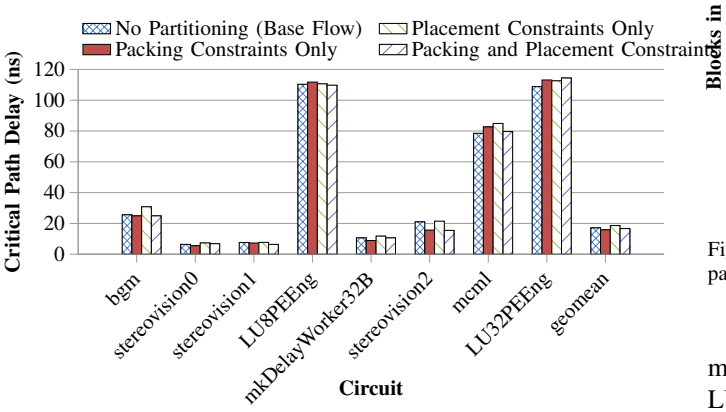


Fig. 17. Critical path delay achieved by VPR for each partitioning flow.

On average, the critical path delay did not vary significantly between the different partitioning CAD flows, but it appears that partitioning before packing may slightly improve circuit speed, while enforcing these partitioning constraints during placement may slightly reduce circuit speed.

Considering both routability and delay, the best CAD flow used the partitioning result to constrain both packing and placement, and we use that flow in the remainder of this paper.

5) % wires cut: To see how well the CAD flow works for a range of interposer architectures, we varied the fraction of wires cut and measured the average minimum channel width over the 8 largest VTR circuits. As shown in Figure 18, this new flow is a significant improvement over that of [6] for FPGAs where more than half the wires are cut at interposer boundaries. It is slightly worse than the traditional FPGA CAD flow even for an architecture with 0% of the wires cut, which is a monolithic FPGA, presumably because we have restricted the placer's freedom.

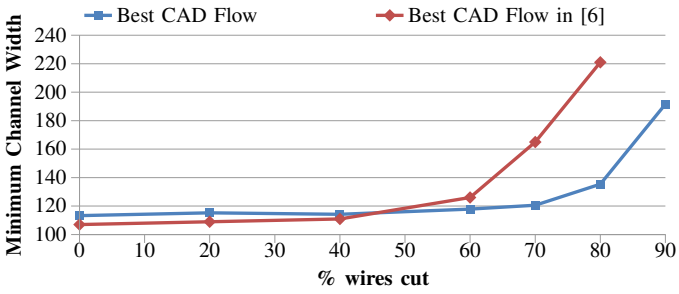


Fig. 18. Impact of partitioning-based CAD flow on routability for a range of architectures.

6) Bloating of the Clustered Netlist due to Partitioning Constraints: The additional constraints on the packer imposed by partitioning may result in a less dense packing. Figure 19 compares the number of complex blocks in the partitioning-constrained and unconstrained clustered netlists.

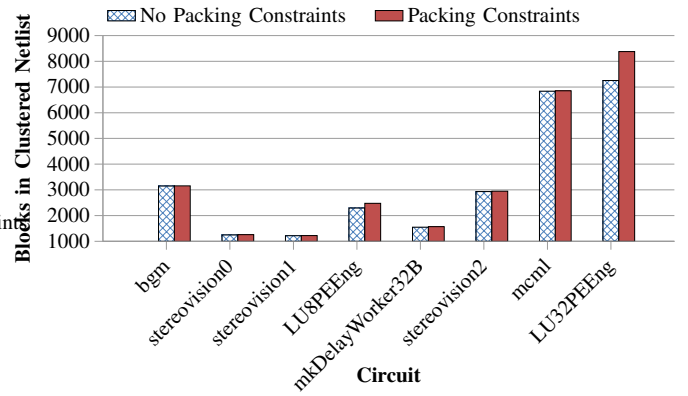


Fig. 19. The number of blocks in the clustered netlist with and without partitioning constraints. This is a measure of packing bloat.

On average, the packing-constrained netlist contained 3.2% more complex blocks than the unconstrained netlist. The LU8PEEng and LU32PEEng circuits were the most affected by this packing bloat. In those circuits, we found that the packing bloat was due almost entirely to poor packing of memory blocks; an interesting area for future work would be to augment the partitioner to understand that some RAM primitives are best kept together in one partition.

VI. Architecture Results

Using the best CAD settings found in Section IV and the best CAD flow described in Section V, we analyze the impact of the key architecture parameters: *% wires cut*, *delay increase*, *number of cuts*, and *interposer-routing interface*. We run all the experiments using the setup detailed in Section IV-C.

A. Routing-Interposer Interface

As described in III-D, we have added three boolean options to VPR that allow us to experiment with different routing-interposer interfaces. Some interface choices provide more flexibility to route signals between different dice and hence help us achieve lower *minimum channel width* and *critical path delay*.

To measure the impact of routing-interposer interface parameters, we set *delay increase*=1ns, *number of cuts=1*, and *% wires cut=80%*; then, we measure the impact of the *bidirectional interposer wires*, *fanin transfer*, and *fanout transfer* parameters in different combinations. Figure 20 summarizes the results of these experiments; all results are normalized to the least flexible routing-interposer interface: one in which 80% of wires end at the interposer and the other 20% cross the interposer with the same directionality and routing switches they would have in a conventional FPGA.

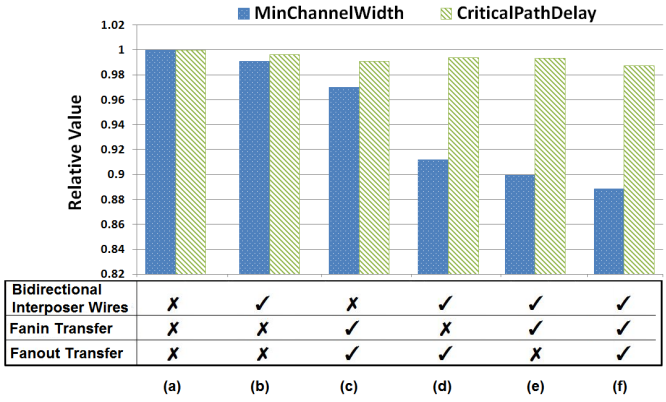


Fig. 20. Impact of routing-interposer interface flexibility on minimum channel width and circuit speed.

Figure 20 shows that using bidirectional interposer wires instead of unidirectional wires (column *b* in Figure 20) improves minimum channel width by 1% and critical path delay by 0.5%. Therefore, while it is beneficial to use bidirectional interposer wires, in isolation they do not dramatically improve performance.

Adding multiplexers in front of interposer wires allows dangling routing wires to connect to interposer wires (*"fanin transfer"*) and adding extra load on the output of the interposer wires (*"fanout transfer"*) allows the interposer wire to drive dangling wires on the other side of the cut. Therefore these modifications provide higher flexibility to better utilize interposer wires. Figure 20 shows that enabling these two options alone without using bidirectional interposer wires (column *c* in Figure 20) reduces minimum channel width by 3% and reduces critical path delay by 1%.

Moreover, better performance can be achieved if either of these two options is used together with bidirectional interposer wires. Enabling *fanout transfer* with bidirectional interposer wires (column *d* in Figure 20) reduces the minimum channel width by 9% and critical path delay by 0.7%. Enabling *fanin transfer* with bidirectional interposer wires (column *e* in Figure 20) has an even stronger impact and reduces the minimum channel width by 10% and critical path delay by 0.7%. This signifies that added flexibility to drive an interposer wire is more important for routability compared to adding flexibility of having an interposer wire drive multiple vertical routing wires on the other side of the cut.

Enabling all of these architecture modifications simultaneously (column *f* in Figure 20) provides the highest level of flexibility for the router and achieves 11.2% reduction in minimum channel width and 1.3% reduction in critical path delay. These models show that interposer-based FPGAs significantly benefit from hardware that provides a flexible routing-interposer interface. It is important both to make the interposer wires easier to access with additional routing switches *and* to make them bidirectional to accommodate asymmetric routing demand.

The *fanin transfer* and *fanout transfer* options ensure that no vertical routing wires are dangling at the cutline; hence, each vertical routing wire either has exactly *one* interposer wire driving it or it drives a mux feeding exactly *one* interposer wire. As a final experiment, we try to add additional fanins and fanouts to interposer wires, so that each vertical routing wire can *potentially* drive multiple interposer wires, or be driven by one of many interposer wires. We found that adding these extra fanouts to interposer wires does not help performance or area, but adding additional fanins to interposer wire muxes helps reduce the minimum channel width while maintaining the same critical path delay. As shown in Figure 21, adding 4 additional inputs to interposer wire muxes reduces the minimum channel width by 1% and adding 8 additional inputs reduces minimum channel width by 1.5%. After that point the performance is no longer limited by the ability to get on an interposer wire and hence we do not see any gains for adding more than 8 additional inputs. Notice that column *0* of Figure 21 is the same as column *f* of Figure 20.

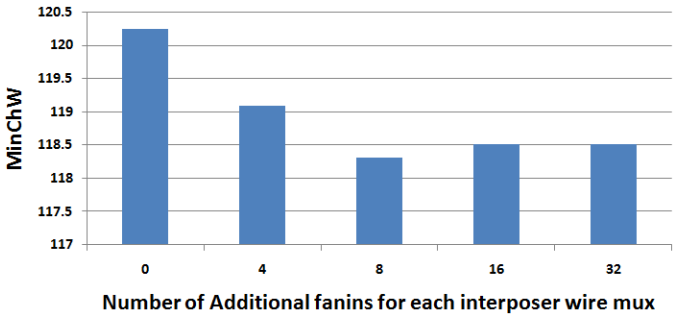


Fig. 21. Adding up to 8 extra fanins to interposer wire muxes helps reduce minimum channel width.

Clearly using bidirectional interposer wires with fanin transfer and fanout transfer is very beneficial, so we use these parameters in all future experiments. Adding additional fanin

beyond transferring the fanin of cut wires is only a small further benefit so we use additional fanin=0 in subsequent sections. Adding these additional multiplexers will have a very small impact on die size as 10,000 muxes per cut would be typical in a large FPGA [7] which is a very small fraction of the several million routing multiplexers in a large FPGA die.

B. Interposer Wiring Capacity (% wires cut)

To analyze the impact of the number of cut wires we ran experiments varying only this parameter while leaving the other parameters constant. We use *number of cuts* $= 1$ (i.e. two dice), *increased delay* $= 1ns$, *bidirectional interposer wires* = *on*, *fanin transfer* = *on*, and *fanout transfer* = *on*.

Figure 22 shows the graph of minimum channel width versus *% wires cut*. It can be noted from this graph that the minimum channel width is essentially constant up to 60% of wires cut, indicating that the CAD flow is able to avoid saturating the interposer routing until that point. When more than 80% of the wires are cut however (i.e. the interposer provides less than 20% of the usual within-die routing capacity), the minimum channel width grows rapidly, indicating that the interposer routing bandwidth has become a limiting factor.

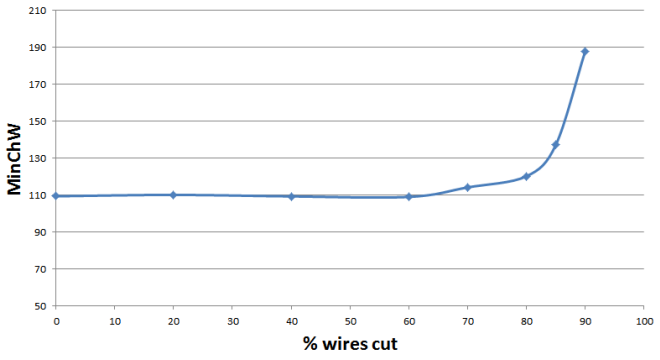


Fig. 22. Minimum channel width vs. *% wires cut* for 2 dice and $1ns$ of *delay increase*.

Figure 23 provides an alternative way to visualize the relationship between interposer routing supply and routability. This figure shows how the geometric average minimum channel width required within the FPGA dice varies as the geometric average of the *absolute* number of wires crossing the interposer in each channel varies, again for 2-die system. When 110 tracks cross the interposer, the interposer channels have the same capacity as the vertical routing channels within each FPGA die. As fewer wires cross the interposer, the channel width required within the FPGA dice increases to compensate for the routing difficulty in crossing the interposer. The increase is gradual as the interposer routing is reduced from 110 tracks per channel to 34 tracks per channel; over this range the routing per channel required in the FPGA dice increases from 110 tracks per channel to 114 tracks per channel. As the routing crossing the interposer is further reduced however, it becomes very difficult to increase the within-die routing sufficiently to compensate. At 18 tracks crossing the interposer channels, for example, the within-die routing must have a channel width of *188* tracks to successfully route the designs.

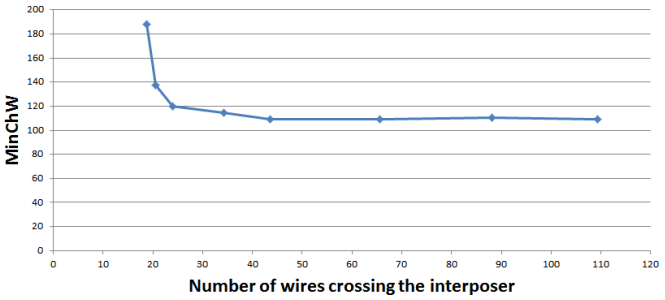


Fig. 23. Geometric mean of required intra-die minimum channel width vs. geometric mean of the number of wires crossing the interposer for 2 dice and $1ns$ of *delay increase*.

Clearly the CAD tools have the ability to trade-off interposer routing for within-die routing over a reasonable range but below a certain level (20% of the original within-die minimum channel width in our experiments) routability becomes almost solely limited by the wiring crossing the interposer and further reduction in interposer routing is not productive.

The critical path delay, depicted in Figure 24, on the other hand, is not strongly influenced by the percentage of wires cut, as the critical path delay at 80% of the wires cut is essentially the same as at 0% wires cut. Note, however, that this same critical path delay is achieved at a much higher channel width when *% wires cut* is greater than 80%.

C. Circuit Speed vs. Interposer Delay

To investigate the impact of the interposer delay (*delay increase*), we keep all other interposer parameters constant and sweep the *delay increase* from 0 to 1.5 *ns*. We use *number of cuts* $= 1$ (i.e. two dice), *bidirectional interposer wires* = *on*, *fanin transfer* = *on*, and *fanout transfer* = *on*.

Figure 24 shows critical path delay versus *% wires cut* for 4 different values of *delay increase*. The penalty in critical path delay is significant, ranging between 1 and 3.7 times the interposer *delay increase*, when compared to the case where the interposer adds no delay. Note that the 0% wires cut with a 0 ns *delay increase* in Figure 24 corresponds to a traditional monolithic FPGA. The speed of an interposer-based FPGA is strongly correlated to *delay increase*: a $0.5ns$ interposer delay

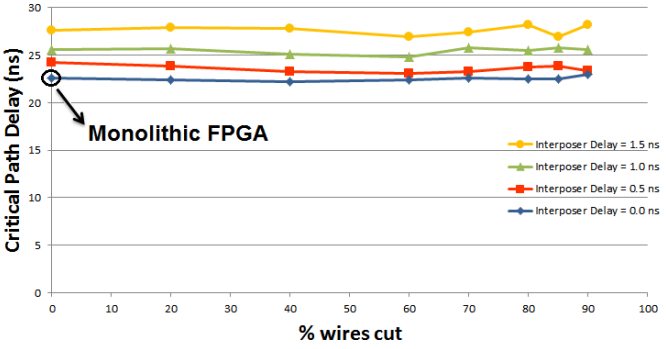


Fig. 24. Critical path delay vs. *% wires cut* for 2 dice and 0.0, 0.5, 1.0 and $1.5ns$ of *delay increase*.

increases the critical path delay by 4%, while a $1ns$ interposer delay increases critical path delay by approximately 12% vs. a monolithic FPGA. But once again, the critical path delay shows little correlation to the *% wires cut*.

D. Impact of Number of Dice

To examine the impact of the number of dice used to construct an interposer-based FPGA, we compare the minimum channel width for FPGA systems composed of four dice and two dice, respectively, while varying the fraction of wires cut at the interposer boundaries. We used the best CAD flow from Section V-A4 and the same interposer routing architecture parameters as Section VI-B. As Figure 25 shows, four-die systems perform almost as well in routability as two-die systems until the *% wires cut* exceeds 80%, at which point routability is worse for the four-die system.

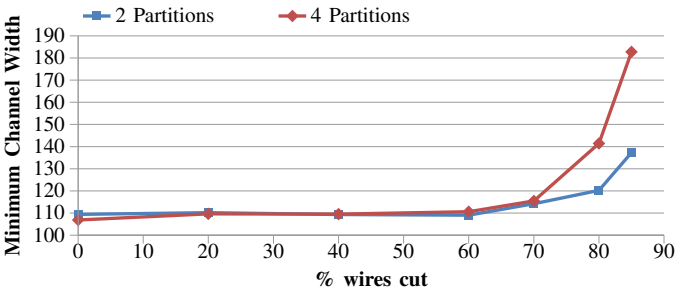


Fig. 25. Impact of number of dice on routability.

Systems with more dice are slower; on average the critical path delay is 40% higher for a four-die system than a two-die system. This speed difference is not significantly affected by the *% wires cut*.

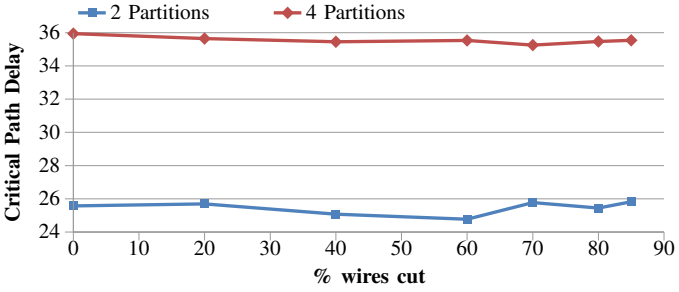


Fig. 26. Impact of number of dice on critical path delay.

E. Impact of Aspect Ratio

It is easier to manufacture silicon dice that have an aspect ratio near 1 (i.e. square). Commercial FPGAs are laid out with identical columns, and hence combine multiple identical FPGA dice in *one dimension* on an interposer [5]. As Figure 27 shows, we can make *either* the FPGA dice square, or the interposer square, but not both. Table I compares the routability of these two options for the four-die case: a square interposer (four rectangular dice) or a rectangular interposer (four square dice.) We used the best CAD flow from Section V-A4 and the same routing architecture parameters as Section VI-B.

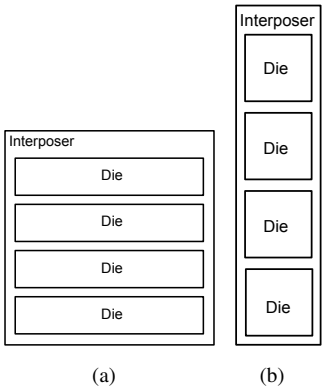


Fig. 27. Devices with (a) a square interposer and (b) square dice.

TABLE I
SQUARE INTERPOSER AND SQUARE DICE MINIMUM CHANNEL WIDTHS

	Square Interposer	Square Dice
bgm	102	306
stereovision0	86	256
stereovision1	166	256
LU8PEEng	156	126
mkDelayWorker32B	96	194
mcml	196	496
LU32PEEng	256	264
geomean	141	252

The square interposer configuration performs significantly better than the square dice configuration. A square interposer results in cutlines that are closer together, and hence more logic blocks are near a cutline. On the other hand, a square interposer (composed of rectangular FPGA dice) has more vertical routing channels, giving it a higher absolute number of wires between FPGA dice. The second effect is the more important one, leading to much better routability for a square interposer.

VII. Conclusion

We have extended VPR to target interposer-based multi-FPGA systems. We found that by modifying VPR's placement cost function we can improve routability, while simultaneously improving speed. Modifications to the VPR router lookahead function are crucial to keep compile time reasonable and yield a 31× speed-up. Incorporating partitioning into the CAD flow is very beneficial. Our best partitioning flow converts nets (hyperedges) to a star graph model, uses Metis to partition the resulting circuit graph, and forces VPR's packer to respect the Metis partitioning to FPGA dice. Taken together, these CAD changes reduce the signal wires that must cross the interposer by an average of 47% while simultaneously improving circuit speed by 13% for realistic interposer architectures.

We defined four key architecture parameters for interposer-based FPGAs, and used this extended VPR to analyze their impact on minimum channel width and critical path delay. We find that adding a moderate amount of switching flexibility at the FPGA - interposer boundary has negligible area cost but

reduces the required minimum channel width by 13%. We also find that the multi-die system has good routability so long as the interposer provides sufficient routing bandwidth, and this routing bandwidth can be much less than one would find in a monolithic FPGA. Specifically, if the interposer provides 40% or more of the channel width required by the monolithic FPGA (an average of 45 wires per channel across our benchmark suite) the routability of the interposer-based FPGA is equivalent to that of a monolithic one. As the interposer signal bandwidth becomes more constrained, routability is impacted but even an interposer providing only 20% of the monolithic FPGA routing channel bandwidth (22 wires in our experiments) only needs a 10% increase in the within-FPGA-die routing channel width to compensate.

The *critical path delay* is not strongly influenced by the *% wires cut* but is strongly influenced by the interposer delay and the number of cuts. On average we find that a 1 ns interposer delay causes an 11% delay increase in a two-die system vs. a monolithic FPGA. Increasing the number of dice in an interposer-based FPGA only moderately impacts the *minimum channel width*, but does lead to a larger *critical path delay*. The aspect ratio of the interposer-based FPGA is an important architectural parameter; a square interposer containing rectangular FPGA dice has the best routability.

Overall we find the results for interposer-based FPGAs to be very positive. While an interposer can only provide a fraction of the signal count between dice that one would have across cutline within a die, the system still has excellent routability so long as the interposer signal count is not pushed to very low values; below 20% of the within-die routing in our experiments. We have also found that modifications to the CAD flow and routing architecture to exploit interposer-based FPGAs have been very productive. As this is a new area we believe more routability and timing enhancements are likely with further research; for example, a timing-driven partitioner may reduce the delay impact of the interposer.

ACKNOWLEDGMENT

This work was supported by Brazil's Ciência sem Fronteiras scholarship and the NSERC/Altera Industrial Research Chair in Programmable Silicon. Some computations were performed on the GPC supercomputer at the SciNet HPC Consortium.

REFERENCES

[1] K. Namhoon, D. Wu, D. Kim, A. Rahman, and P. Wu, "Interposer Design Optimization for High Frequency Signal Transmission in Passive and Active Interposer using Through Silicon Via (TSV)," in *IEEE Electronic Components and Technology Conf.*, 2011, pp. 1160–1167.

[2] P. Garrou, M. Koyanagi, and P. Ramm, *Handbook of 3D Integration: Volume 3 - 3D Process Technology*. John Wiley and Sons, 2014.

[3] M. Alexander, J. Cohoon, J. Colflesh, J. Karro, and G. Robins, "Three-Dimensional Field-Programmable Gate Arrays," in *IEEE Int. ASIC Conf. and Exhibit*, 1995, pp. 253–256.

[4] M. L., A. El Gamal, Y. Lu, and S. Wong, "Performance Benefits of Monolithically Stacked 3-D FPGA," *IEEE Trans. on CAD*, vol. 26, no. 2, pp. 216–229, 2007.

[5] R. Chaware, K. Nagarajan, and S. Ramalingam, "Assembly and Reliability Challenges in 3D Integration of 28nm FPGA Die on a Large High Density 65nm Passive Interposer," in *IEEE Electronic Components and Technology Conf.*, 2012, pp. 279–283.

[6] A. Hahn Pereira and V. Betz, "CAD and Architecture for Interposer-Based Multi-FPGA Systems," in *ACM Int. Symp. on FPGAs*, 2014, pp. 75–84.

[7] Xilinx, "7 Series FPGA Overview," *www.xilinx.com*, 2013. [Online]. Available: www.xilinx.com

[8] Altera, "Stratix V Device Overview," *www.altera.com*, 2013. [Online]. Available: www.altera.com

[9] Xilinx, "UltraScale Architecture and Product Overview," *www.xilinx.com*, 2014. [Online]. Available: www.xilinx.com

[10] J. Cunningham, "The Use and Evaluation of Yield Models in Integrated Circuit Manufacturing," *IEEE Trans. on Semiconductor Manufacturing*, vol. 3, no. 2, pp. 60–71, 1990.

[11] Xilinx, "Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency," *www.xilinx.com*, 2012. [Online]. Available: www.xilinx.com

[12] J. Luu *et al.*, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 7, no. 2, p. 6, 2014.

[13] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Int. Workshop on Field-Programmable Logic and Applications*, 1997, pp. 213–222.

[14] G. Lemieux, E. Lee, M. Tom, and A. Yu, "Directional and Single-Driver Wires in FPGA Interconnect," in *IEEE Int. Conf. on Field Programmable Technology*, 2004, pp. 41–48.

[15] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault *et al.*, "The Stratix II logic and routing architecture," in *ACM Int. Symp. on FPGAs*, 2005, pp. 14–20.

[16] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic Publishers, 1999.

[17] A. Marquardt, V. Betz, and J. Rose, "Timing-Driven Placement for FPGAs," in *ACM Int. Symp. on FPGAs*, 2000, pp. 203–213.

[18] C. Cheng, "RISA: Accurate and Efficient Placement Routability Modeling," in *IEEE/ACM Int. Conf. on CAD*, 1994, pp. 690–695.

[19] J. Rose, J. Luu, C. Yu, O. Densmore, J. Goeders, A. Somerville, K. Kent, P. Jamieson, and J. Anderson, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," in *ACM Int. Symposium on FPGAs*, 2012, pp. 77–86.

[20] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

[21] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Domain," *IEEE Trans. on VLSI*, vol. 7, no. 1, pp. 69–79, 1999.

[22] G. Karypis, "A Software package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices," *University of Minnesota*, 2013.

[23] W. Feng, J. Greene, K. Vorwerk, V. Pevzner, and A. Kundu, "Rent's Rule Based FPGA Packing for Routability Optimization," in *ACM Int. Symp. on FPGAs*, 2014, pp. 31–34.