FPGA Placement by Graph Isomorphism

Hossein Omidian Savarbaghi
Dept. of Computer Eng.
Islamic Azad University
Science and research branch
Tehran, Iran
h_omidian_sa@yahoo.com

Kia Bazargan
Dept. of Elec. and Computer Eng.
University of Minnesota
200 Union St SE
Minneapolis, MN 55455
kia@ece.umn.edu

ABSTRACT
FPGA placement and routing are still challenging problems. Given the increased diversity of logic and routing resources on FPGA chips, it seems appropriate to tackle the placement problem as a mapping between the nodes and edges in a circuit graph to compatible resources in the architecture graph. We explore utilizing graph isomorphism algorithms to perform FPGA placement. We use a hierarchical approach in which the circuit and architecture graphs are simultaneously clustered to reduce the size of the search space, and then a novel reductive graph product method is used to solve the isomorphism problem. The graph product algorithm is called reductive as it eliminates a linear number of candidates at every step of the search process, reducing the number of candidate nodes by approximately 1/3. Compared to the annealing-based placement tool VPR 5.0, we achieve approximately 40% improvement in placement runtime, while improving the critical path delay by about 7% and wire length by 5%, while demanding 1.3% more channels on average.

Categories and Subject Descriptors
J.6 [COMPUTER-AIDED ENGINEERING]: Computer-aided design.

B.7.2 [INTEGRATED CIRCUITS]: Design Aids- *Placement and routing.*

General Terms
Algorithms, Design, Theory.

Keywords
Placement, graph isomorphism, graph product, clustering.

1. INTRODUCTION
The FPGA placement problem has been studied for a few decades. The solutions to this problem include meta-heuristics (e.g., annealing [1], ant colony [2]), hierarchical [3], and parallel implementations [4],. The placement (and routing) problem is becoming more challenging as the number of resources on the chip are growing exponentially [5] and FPGAs contain increasingly heterogeneous collection of resources [6][7] (LUTs, memory blocks of different sizes, DSP blocks, serial I/O, multipliers, and even embedded processor cores).

ACM/SIGDA International Symposium on Field Programmable Gate Array (FPGA 2011), Monterey California

Placement and routing are in essence graph mapping problems: from the circuit graph to the architecture graph, matching the types of circuit nodes (e.g., memory or multiplication operations) to compatible types in the architecture graph (e.g. memory can be mapped either to embedded memory blocks or collection of LUTs. Similarly, a multiplication operation in the circuit graph can be mapped either to an embedded multiplier unit or a soft multiplier mapped to the LUTs). The same is true for routing resources: timing critical nets have to be mapped to faster routing resources.

Graph Isomorphism algorithms are potentially good candidates for simultaneously solving the placement and routing problems, but their success has been limited primarily due to their high time complexity. We utilize the high degree of symmetry in the architecture to significantly cut on the runtime of graph isomorphism algorithms. We employ clustering, as well as a novel reductive graph multiplication algorithm to solve the graph isomorphism problem efficiently. Compared to the well-known annealing-based algorithm VPR 5.0 [1], we achieve about approximately 40% speedup while improving delay (7%) and wire length (5%) and slightly worsening channel width (1.3%).

The rest of the paper is as follows: Section 2 gives a high-level description of our algorithm. Section 3 covers the background material for graph isomorphism and clustering. Details of our approach is described in Section 4. Experimental results are presented in Section 5, followed by conclusions and future directions in Section 6.

2. OVERVIEW OF THE APPROACH
Our approach uses a graph isomorphism algorithm as its core, but uses two main techniques to speed up the algorithm. The first technique is to adopt a hierarchical methodology to reduce the search space. The challenge is to maintain enough architectural details to achieve a high-quality solution at every level of the hierarchy. The second technique is to use the inherent abundance of symmetry in the architecture graph to trim the search space significantly.

Figure 1 shows the overall flow of our algorithm, which uses a “V” cycle [8][9] of clustering and local bottom-up placement, followed by unclustering and final top-down placement in a hierarchical fashion. The circuit graph (top-left in Figure 1) is first recursively clustered using the method described in [10] until the cluster size limits are reached (a maximum of 9 nodes in each cluster in our case). Then each cluster is mapped to a subset of the architecture graph (e.g., an array of 3x3 logic elements), which essentially means finding a local placement solution for each cluster. The process is repeated in a hierarchical fashion until the number of clusters is small enough (the bottom-most oval in

Figure 1). After that, the unclustering phase begins, in which local placements are packed together and the final coordinates of nodes are assigned.

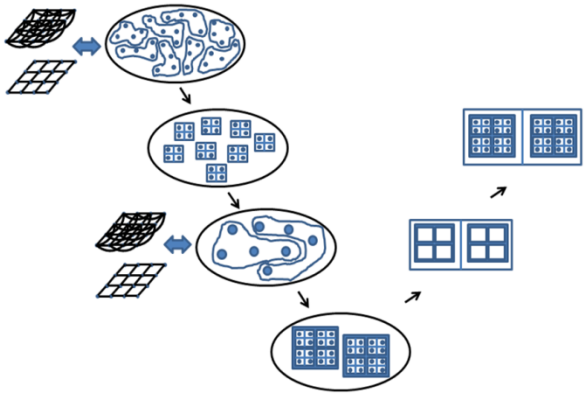

Figure 1. Overall flow of our algorithm

There are five major steps in our algorithm. Phase 1 is initialization. Phases 2 – 4 are repeated hierarchically (arrows going down in Figure 1), followed by Phase 5 (arrows going up):

- *Phase 1: initialization.* In this phase, timing analysis is done on the circuit graph to determine timing critical nets. Furthermore, the architecture graph is pre-processed to generate hierarchical representations of the architecture to be used during the graph isomorphism phase.

- *Phase 2: Clustering*. In this phase, the aim is to cluster highly connected logic elements (LEs) in the circuit, subject to cluster size limits. Clustering results in smaller circuit graph sizes that help maintain reasonable runtimes during the isomorphism phase. The quality of clustering has a major impact on the overall quality of our algorithm.

- *Phase 3: Finding graph isomorphism.* After one level of clustering is done, each cluster represents a small circuit sub graph to be mapped to the corresponding hierarchy of the architecture graph. The best sub graph isomorphism of these two graphs is found in which timing critical edges are mapped to faster routing resources. The principal issue is to find this sub graph isomorphism in the lowest time, otherwise the best sub graph isomorphism could be found by spending a huge amount of time exhaustively considering all cases.

- *Phase 4: Local placement.* After finding the sub graph isomorphism of the circuit and architecture sub graphs, each node of the circuit graph is labeled by a local placement coordinate. Although we do not currently perform additional local optimizations in this phase, our future work includes further refining local placements by repeating static timing analysis and repeating the local placement phase.

- [Phases 2—4 are repeated recursively until the number of clusters is small enough. Each cluster in Phase 4 is now considered as a node to be clustered in Phase 2]

- *Phase 5: Unclustering and final placement.* After the bottom-up placement of the circuit is done, a top-down placement phase begins that stitches together the local placements, assigning final coordinates to all nodes. Although we do not currently apply further optimizations during the top-down placement phase, our future work includes exploring refinements such as considering the rotations or mirror images of local placements when stitching them together to build the final placement.

3. PRELIMINARIES

In this section we briefly cover a few of the existing algorithms for graph isomorphism and also graph clustering problems. Our technique employs a novel reductive graph multiplication algorithm to solve the isomorphism problem and Marek-Sadowska's clustering approach [10] to build the hierarchy of the architecture and the circuit graph.

3.1 Graph isomorphism

The graph isomorphism problem has been extensively studied in the past and there are many algorithms that target various classes of the problem. We first give a formal definition of the problem and then list a number of well-known algorithms.

Definition 1: the graph isomorphism problem. Given two graphs $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$, there is an exact isomorphism between the graphs if $|V_1|=|V_2|$ and $|E_1|=|E_2|$ and we can find a mapping $\Omega: V_1 \rightarrow V_2$ that maps every node of G_1 to a unique node of G_2 such that $\forall(u_1,v_1)\in E_1$, $(\Omega(u_1), \Omega(v_1)) \in E_2$ and $\forall(u_2,v_2)\in E_2$, $(\Omega^{-1}(u_2), \Omega^{-1}(v_2)) \in E_1$. In other words, there is isomorphism between two graphs if there is a one-to-one correspondence between the nodes and the edges of the two graphs.

The complete isomorphism problem has limited applications. In many problems, it is not important to find an exact match between two graphs, but rather to find an isomorphism between a graph and a *sub-graph* of the second graph.

Definition 2: the partial graph isomorphism problem. Given two graphs $G_1(V_1,E_1)$ and $G_2(V_2,E_2)$, there is a partial graph isomorphism between the two graphs if G_1 contains a sub-graph $G_3(V_3,E_3)$ that is isomorphic to G_2. In other words for a graph $G_1(V_1,E_1)$, $\exists$ set $V_3 \subseteq V_1$ and $E_3 \subseteq E_1$ such that $|V_3|=|V_2|$ and $|E_3| \leq |E_2|$ *and there is a mapping* $\Omega: V_3 \rightarrow V_2$ *that maps every node of* G_3 *to a unique node of* G_2 *such that* $\forall(u_3,v_3)\in E_3$, $(\Omega(u_3), \Omega(v_3)) \in E_2$ *(but not necessarily* $\forall(u_2,v_2)\in E_2$, $(\Omega^{-1}(u_2), \Omega^{-1}(v_2)) \in E_3$*)*

Intuitively, for our placement application we are interested in a sub-set of the architecture graph to which we can map every node and edge of the circuit graph (but not necessarily map every edge in the architecture graph to an edge in the circuit graph).

3.1.1 Nauty

Generally speaking, graph isomorphism algorithms suffer from long runtimes and are not scalable. One of the best open-source sub graph isomorphism implementations is Nauty (**N**o **aut**omorphisms, **y**es?). Nauty is "*a set of procedures for determining the automorphism group of a vertex-coloured graph. It provides this information in the form of a set of generators, the size of the group, and the orbits of the group. It is also able to produce a canonically labeled isomorph of the graph, to assist in isomorphism testing*" [11]. The main features of the algorithm are explained in [11], although the authors have made significant improvements in the latest implementation.

In our initial implementations, we used Nauty as our isomorphism engine, but its runtime was high and hence we explored faster

methods that utilize the high degree of symmetry in the FPGA architecture graph. Graph product methods for finding sub graph isomorphism between two graphs are of similar runtimes to Nauty, but are much more suitable for utilizing the symmetries and cutting on the runtime.

3.1.2 Graph Product

The product of two graphs is a graph that determines the degree of similarity between all pairs of nodes between two graphs. Nodes of the product graph of $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, are ordered pairs (v, w) where $v \in V_1$ and $w \in V_2$ and are indexed based on original graph index. There is an edge between (v_a, w_c) and (v_b, w_d) in the product graph if and only if either of the following two conditions are met: (1) there is an edge between v_a and v_b in G_1 and there is an edge between w_c and w_d in G_2, or (2) there is no edge between v_a and v_b in G_1 and no edge between w_c and w_d in G_2. More formally, the product graph of graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ is graph $G(V, E)$ where its node set is $\{(v, w) \in V_1 \times V_2,\ \{(v, w) | v \in V_1, w \in V_2\}\}$ and its edge set is

$$E = \left\{((v_a, w_c), (v_b, w_d)) \in V_1 \times V_2 \middle| \begin{matrix} v_a \neq v_b, w_c \neq w_d \\ ((v_a, v_b) \in E_1, (w_c, w_d) \in E_2) \end{matrix}\right\}$$

$$\bigcup \left\{((v_a, w_c), (v_b, w_d)) \in V_1 \times V_2 \middle| \begin{matrix} v_a \neq v_b, w_c \neq w_d \\ (v_a, v_b) \notin E_1, (w_c, w_d) \notin E_2) \end{matrix}\right\}$$

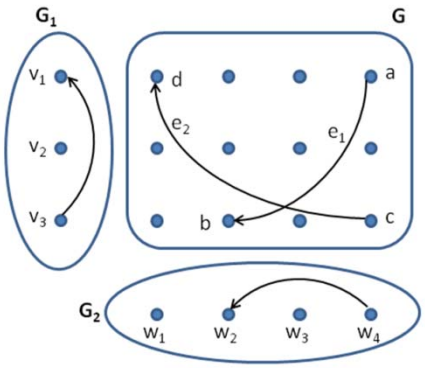

Figure 2. Product graph: graph G is the product of graphs G_1 and G_2.

As depicted in Figure 2, in the product graph there is an edge labeled e_1 between nodes a and b, because there is an edge between v_1 and v_3 in E_1 and also there is an edge between w_2 and w_4 in E_2. On the other hand, there is an edge between nodes c and d in the product graph labeled e_2, because there is no edge between v_1 and v_2 in E_1, and none between w_4 and w_1 in E_2.

The product graph can be used in finding sub graph isomorphisms between two graphs (Section 4.3). We later show in Section 4 how we utilize graph products to perform local placement.

3.2 Clustering

We tried two clustering approaches: hMETIS [12] and Marek-Sadowska's clustering method [10]. The latter performed better in our application.

3.2.1 Partitioning using hMETIS

hMETIS [12] is a well-known partitioning package for partitioning large hypergraphs. It uses a multi-level approach of clustering and unclustering nodes, and handles hyper edges efficiently. Its quality is one of the best for minimizing the cut cost across partitions.

Even though hMetis works well in top-down partitioning-based placement algorithms that minimize wire length [8], but in our isomorphism approach, it did not yield high quality results for minimizing timing critical net lengths. To address this issue, we tried applying static timing analysis and delay budget assignment [14] [15] to give higher weights to timing critical nets, but even with timing weights, hMetis did not fare as well as Marek-Sadowska's approach [10] in our framework.

3.2.2 Clustering using Marek-Sadowska's work [10]:

We briefly describe the clustering method used in [10]. We used this clustering algorithm in our approach. A circuit is represented as a weighted digraph G(V,H,w), where V is the set of nodes (corresponding to LEs in an FPGA), H is the set of hyperedges between nodes and w(e) is a positive hyperedge weight assigned for each $e \in H$. Hyperedge weights are inversely proportional to the cardinality of the hyperedge therefore smaller nets will have higher weights.

Also, to each node a *degree* will be assigned which equals the

The *Connectivity factor (c)* of an LE is defined as:

$$c = \frac{Separation}{degree\ 2}$$

where the degree of the node is the number of nets connected to that LE, and the *separation* of an LE is the number of all terminals connected to it by all its incident edges. Needless to say, smaller values of *c* signify LEs with more LEs in their neighborhood.

The very first step in clustering is to determine the *c factor* for all unclustered LEs and select an LE with the higher *degree* and lower *c* as a seed for a new cluster. This seed will absorb other nodes to the cluster. Then, a *Gain* value is assigned to each unclustered node. The gain is defined as:

$$G(X, C, x) = 2nw(x) \times (1 - \alpha_x)$$

Where X is the candidate nodes set, C is a currently open cluster, x is the connected net set, α_x is the number of pins of net x that are already inside C, n is the cluster size and w(x) is the weight of net x (w(x)=2/r where r is the number of pins on x).

We set the maximum number of nodes inside a cluster to a *limit*, which we choose to be the number of LEs in a hierarchy level of the architecture graph (in our case, we chose the limit to be 9, which corresponds to a subset of 3x3 LEs in the architecture graph).

4. OUR APPROACH

The outline of our approach was described in Section 2. In this section we provide a more detailed description of our algorithms.

Section 4.1 lists the main pseudo-code of our approach. In the first phase (lines 6 to 8), we perform static timing analysis and assign timing criticalities to edges in the circuit graph. We also build

hierarchical sub graphs of the architecture (Section 4.2 provides more details). In Phase 2 (lines 10 to 20) we continue clustering until all nodes are clustered. For clustering, first we have to determine the *degree* and *connectivity c* values (Section 3.2.2) for each node (line 10) and based on these values, choose the best candidate among unclustered nodes to act as the seed for the new cluster (line 13). Then the seed will absorb the best unclustered nodes based on the Gain function. After absorbing each node to the cluster, in order not to use this node in the following steps, its *is_clustered* variable is set to 1 (line 17). The clustering process continues while the number of absorbed nodes is no more than *cluster_limit* (*cluster_limit* is chosen based on the architecture graph). At the end of this phase, we will have as many clusters as the value of *cluster_number* variable. After clusters are formed, we perform phases 2 and 3 for each cluster independently (lines 23 to 32).

In Phase 3, we take each cluster and find a sub-graph isomorphism between the nodes in the cluster and the corresponding hierarchy level architecture sub-graph. We provide more details on modeling the architecture graph (Section 4.2) that allow prioritizing circuit edges and mapping timing critical edges to faster routing resources in the architecture graph. More detailed explanation of the isomorphism process itself is provided in sections 0 and 4.4).

After finding sub-graph isomorphism, a mapping between the circuit graph and the architecture graph is found. As a resultfor each cluster, local placement of each node in the cluster is determined (line 31).

After local placement of all clusters in Phase 4, a high level graph is built for top-down placement. To do so, each cluster is considered as a node and edges between clusters become graph edges. It is obvious that internal edges in each cluster will be omitted after considering a cluster as a node (line 33). Then this graph is considered as a new circuit graph (line 34) and the steps detailed above are repeated. *Hierarchical_number* variable saves the number of hierarchical levels (line 35). We continue until the whole graph is considered as one cluster (*number_of_nodes* <= *cluster_limit)*. In Phase 5, we perform the final placement process (lines 37 to 39), in which we start from the highest level and record its final placement. Then we go down each level and adjust node coordinates to combine their local placement with the coordinates from the higher level final placement (line 38). We repeat this for all hierarchical levels.

4.1 Main Algorithm Pseudo-Code

```
1   Input:
2       GRAPH Gc(Vc,Ec)          // Circuit graph
3       GRAPH GA(VA,EA)          // Architecture graph
4   Output:
5       Mapping Π:Gc→GA

6   // Phase 1: initialization
7   Perform timing analysis and assign edge criticalities to ckt
8   Build hierarchical levels of the architecture (Sec. 4.2)
9   While number_of_nodes > cluster_limit
        // Phase 2: Clustering
10      Find degree and C for each node
11      While there are unclustered nodes
12          Allocate Cluster[j] in memory
13          Choose best seed candidate, add to Cluster[j]
14          For i = 1 to cluster_limit do
15              Temp_node ← node with best (Cluster[j], Gc)
                            cost //best node for absorption
16              Cluster[j] ← Cluster[j] ∪ Temp_node
17              Temp_node.is_clustered ← 1
18          End for
19          j++
20      End while
21      Cluster_number ← j
22      /* Phase 3: Find sub graph isomorphism for each
         cluster*/
23      For i = 1 to cluster_number do
            // create product graph P(Vp,Ep)
24          For j = 1 to classes_of_arch_graph_number do
25              P ← Product_graphs (cluster[i] ,
                                    arch_graph_class[j])
26              Find sub graph isomorphism
27              if sub-graph isomorphism found
28                  Break
29              End if
30          End for
        // Phase 4: Place locally
31      Place locally nodes of cluster[i]
32      End for
        // Phase 5-a: Create high level graph
33      Number_of_nodes ← Ceate high level graph
34      Circuit graph ← high level graph
35      Hierarchical_number++
36  End while  // started on Line 6
    // Phase 5-b: Finalize Placement
37  For i = 1 to Hierarchical_number do
38      Use hierarchical node's coordinate to adjust lower level
            coordinates
39  End for
```

4.2 Modeling the Architecture Graph

A challenging problem when dealing with placement instances is how to map timing critical nets to faster routing resources[1]. Methods like annealing try to use intermediate cost functions such as sum of weighted net lengths that represent the delay incurred on a net when its terminals are placed. For example TVPR uses a lookup table that stores the best-case delay incurred if a terminal of a net is placed at the (0,0) coordinate and the other terminal is placed at (x,y). The assignment of critical nets to resources is left to the annealing process.

In our approach, we first discretize the range of net criticalities, and try to map the most critical nets to the fastest routing resources first. Then, unassigned nets are tried with the next "class" of routing resources.

[1] Assignment of "nets" to "routing resources" is done at the routing step. However, assigning terminals of a net to locations on the FPGA implicitly affects the net to routing segment assignment during routing. So in this context, when we talk about assigning nets to segments during placement, we are really talking about placing its terminals with potential routing resource assignment in mind.

As explained in previous sections, for each hierarchy level in the V-cycle (Figure 1), an architecture sub-graph must be built. Figure 3 shows one such graph. As can be seen in the figure, several "class" graphs are built based on the routing resources in the FPGA architecture. Class 1 uses only the fastest routing resources as edges in the architecture sub-graph (e.g., single-segment routing resources if they are the fastest). Class 2 sub-graph contains Class 1 edges in addition to the next class of routing resources (e.g., double lines if their delay is the next best after single lines). This will continue until all nets are represented in the last class graph.

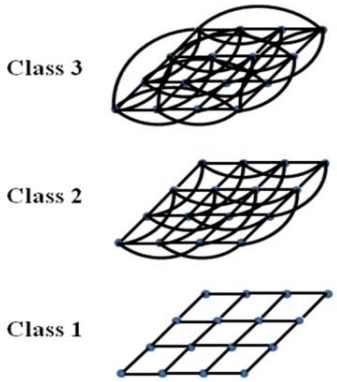

Figure 3. Modeling connections in an architecture sub-graph

To find a sub-graph isomorph of the architecture graph to match the circuit sub-graph, we first run the isomorphism algorithm on Class 1 sub-graph. If we cannot find a solution in the first class, we try the next class (lines 24 to 30 in Section 4.1). Finding sub-graph isomorphism (line 26) is explained in more details in sections 0 and 4.4.

4.3 Reductive Graph Product

The general procedure for building the product of two graphs was described in Section 3.1.2. We noted earlier that finding sub-graph isomorphism using graph products could be time consuming. In this section we explain how we can utilize the high degree of symmetry in the architecture graph to prune out many similar solutions and hence speedup the isomorphism algorithm significantly.

After multiplying two graphs, product graph nodes must be visited to find nodes with higher number of connections. Such nodes indicate that the corresponding nodes in the two graphs have similar connectivity patterns, and hence are good candidates for mapping. A collection of such highly similar nodes is the sub-graph isomorphism between the two graphs. This will be done by weighting each vertex of the product graph. For example, in Figure 4-d, the four solid red circles in P indicate that an isomorphism is found that maps v1 →w3, v2 →w4, v3 → w5, and v4 → w2. We call the collection of the red circles an "isomorphism set". To avoid cluttering the figure, edges are not shown.

We use two heuristics to significantly speed up the search in the product graph to find a collection of nodes that determine the sub-graph isomorphism between the two graphs: (1) eliminating rows and columns, and (2) starting the search from the clustering seed nodes. Below we explain each heuristic in more details.

- *Eliminating rows / columns*: We note that neither of the circuit or architecture graphs contain self-edges (i.e., edges that connect a node to itself). General graph product algorithms have to handle such cases but fortunately we do not. As a result, when looking for candidate nodes in the product graph to build the "isomorphism set", once we choose a node we can eliminate all nodes in that row and column. This process is shown in Figure 4 in which eliminated nodes are shown using dotted circles.
- *Starting from the seed node*: During clustering, one node is always selected as the seed node to absorb neighboring nodes to the cluster. The seed nodes have a higher connectivity compared to other nodes. As a result, we can assure that in multiplying two graphs with appropriate weighting, there are always some nodes with higher weights. Choosing such nodes first has two major advantages: (1) seed nodes are usually connected to the most critical nets, so by finding a match for them first, we indirectly ensure the most critical nets in the current class are assigned first, and (2), given that seed nodes have high degrees, by choosing their corresponding node in the architecture graph first, we increase the likelihood of finding a legal solution for the isomorphism problem in lower classes of the architecture graph.

.

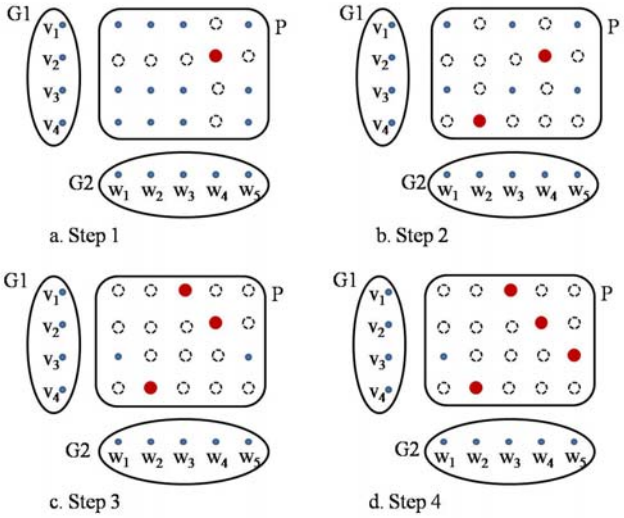

Figure 4. Finding an isomorphism using the product graph

The number of candidate nodes in the original graph product algorithm is $O(V^3)$, where $V=max\{V1,V2\}$ because in each of the V iterations, we have to look for the best weight among V^2 nodes[2]. Our row/column elimination technique reduces the time constant by 1/3, even though it does not change the asymptotic time complexity. The reason for the reduction is that in the first iteration we have V^2 candidate nodes, then by eliminating a row and column we have $(V-1)^2$ candidates and so on. The series $1^2+2^2+\ldots+V^2$ evaluates to approximately $1/3V^3$. For small values of V, which happens to be our case (e.g., V=9, which corresponds

[2] For each candidate node, we have to find the highest weight node. This can be done using heap trees in O(log V) in each step.

to an architecture sub-graph of 3x3 LEs), the time constant has a significant impact.

4.4 Sub-graph Isomorphism Pseudo-Code

1. Input:
2. GRAPH $G_1(V_1,E_1)$ //First Graph
3. GRAPH $G_1(V_1,E_1)$ //Second Graph
4. Output:
5. GRAPH $G_3(V_3,E_3)$ //Sub-graph Isomorphism
6. $P(V_p, E_p, W_p) \leftarrow$ Product_graphs(G_1 , G_2): // W_p weight
7. $V_p = \{(v,w) | v \in V_1 , w \in V_2\}$
8. $E_p = \{((v_a,v_b)(w_a,w_b)) | v_a,v_b \in V_2 , w_a,w_b \in V_2\}$
9. $W_p = K_p$ * Number of edges related to each node
10. For $\forall v \in V_p$
11. Temp_node ← Chose best node(P)
12. Temp.visited ← 1
13. mark all nodes in this row and column as visited
14. End for

5. EXPERIMENTAL RESULTS

We placed and routed the 20 MCNC circuit benchmarks of the VPR package and 3 of the largest benchmarks of the Altera QUIP tool set (oc_web_dma, oc_mem_ctrl and oc_des_des3pref). The results are shown in the following table. VPR refers to VPR 5.0 and ISO refers to our isomorphism algorithm.

The results show that critical path and wire length are better in our approach on average, but that result should be taken with a grain of salt, as in most cases our results are slightly worse, but the overall average is tilted due to a number of cases that show significant improvement (e.g., apex4, exp5p, ex1010, s298).

NOTE: As of now, our code has a bug that does not allow us to place the I/O blocks. As a result, the runtime numbers reported in the column labeled "ISO (w/o IO)" are slightly deflated (VPR numbers do include I/O blocks). To address this issue and provide a more realistic comparison, we analyzed the ISO runtime trend as a function of the CLB count. If we draw a scatter plot that shows the CLB count on the x-axis and the runtime on the y-axis, we notice that the runtime is approximately a linear function of the number of CLBs (this only applies to the 20 MCNC benchmarks: the isomorphism algorithm significantly outperforms VPR on the three large QUIP benchmarks, and hence were set aside as outliers). The scatter plot of the 20 MCNC benchmarks is shown in Figure 5 (last page). We used the trend line to inflate no-I/O runtime numbers to get the numbers in the column labeled "ISO (inflated)".

Table 1. Comparison of ISO with VPR

Circuit	CLB	Net	Runtime (ms)			Critical Path (x10-8)		Max Net Length		Max Ch Width	
			VPR(w/IO)	ISO (w/o IO)	ISO (inflated)	VPR	ISO	VPR	ISO	VPR	ISO
alu4	1522	1536	100187	57134	55172	7.92	7.55	1795	1839	19	17
apex2	1878	1916	129984	73481	82438	7.44	7.47	1609	1661	20	20
apex4	1262	1271	136328	76694	36703	13.82	8.24	865	881	20	20
bigkey	1707	1936	110796	62255	97998	4.76	4.86	3417	3524	13	13
clma	8383	8445	959574	555001	562906	15.05	15.62	9170	5513	23	23
des	1591	1847	104016	61386	95017	5.76	5.86	2579	2698	16	15
diffeq	1497	1561	81953	47935	59244	5.31	5.12	2355	2453	14	14
dsip	1370	1599	86078	48961	73495	6.77	5.34	5746	5341	16	15
elliptic	3604	3735	280594	159608	222768	11.37	11.56	5848	6075	19	19
ex5p	1064	1072	56578	31875	25433	8.42	7.19	1692	1773	20	23
ex1010	4598	4608	379281	214001	278682	15.34	13.49	3094	3189	19	20
frisc	3556	3576	289297	160599	211353	8.67	8.708	4564	3627	23	23
misex3	1397	1411	74204	42447	46519	6.38	6.486	1406	1334	20	22
pdc	4575	4591	425781	241590	279628.01	14.8	14.95	3257	3389	32	32
s298	1931	1935	135156	78342	84038	15.21	9.07	2456	2550	17	17
s38417	6406	6435	760265	434556	418504	8.05	8.18	3738	3850	17	17

s38584.1	6447	6485	807359	467956	436536	5.85	5.99	9114	9277	17	17
seq	1750	1791	138438	79019	75676	6.03	6.17	1408	1439	20	22
spla	3690	3706	437359	247363	215715	12.43	12.29	2566	2175	29	28
tseng	1047	1099	49515	27819	31686	5.16	5.34	1457	1538	13	13
oc_wb _dma	9872	9654	125071	69283	78794	9.37	9.41	1216	1231	28	28
oc_mem _ctrl	8611	8726	112031	59589	70679	8.94	9.01	884	902	20	22
oc_des_ des3perf	38218	38452	140629	87961	88582	15.47	15.59	3317	3386	16	17
Average			**257412**	**147168**	**157717**	**9.49**	**8.84**	**3198**	**3028**	**19.61**	**19.87**
Ratio (ISO / VPR)				**0.57**	0.61		**0.93**		**0.95**		**1.01**

6. CONCLUSIONS AND FUTURE WORK

We presented a hierarchical placement methodology in which a reductive graph product isomorphism algorithm is applied at every level of the hierarchy. We achieved significant speedup over VPR 5.0 while maintaining similar placement qualities.

There are several aspects of the algorithm that we plan to improve in the future. As of now, we only perform placement but do not provide any routing hints to the router. We plan to use isomorphism data during the routing phase so that we can pre-assign preferred routing resources for critical nets, and let the VPR router route the rest of the nets.

An interesting experiment would be to test our algorithm on benchmarks that have heterogeneous blocks such as memory blocks with different sizes, multipliers, etc., and see if such blocks will act as “anchors” in our isomorphism algorithm and significantly reduce runtimes.

Other improvements include considering rotations and mirror images of local placements when building the global placement, and also performing multiple runs of static timing analysis and local placement.

7. REFERENCES

[1] Jason Luu , Ian Kuon , Peter Jamieson , Ted Campbell , Andy Ye , Wei Mark Fang , Jonathan Rose, VPR 5.0: FPGA cad and architecture exploration tools with single-driver routing, heterogeneity and process scaling, Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays, February 22-24, 2009, Monterey, California, USA

[2] Gang Wang, "Ant Colony Metaheuristics for Fundamental Architectural Design Problems", PhD Thesis, University of California Santa Barbara,2007.

[3] Dai H, Zhou Q, Bian JN., "Multilevel optimization for large-scale hierarchical FPGA placement", Journal of Computer Science and Technology, 25(5): 1083-1091 Sept. 2010. DOI 10.1007/s11390-010-1085-4

[4] Cristinel Ababei, "Speeding Up FPGA Placement via Partitioning and Multithreading", International Journal of Reconfigurable Computing, Vol. 2009 (2009)

[5] International Technology Roadmap for Semiconductors (ITRS), http://public.itrs.net/

[6] http://www.xilinx.com

[7] http://www.altera.com

[8] *George Karypis and Vipin Kumar*, “A fast and high quality multilevel scheme for partitioning irregular graphs”, *International Conference on Parallel Processing, pp. 113-122, 1995*.

[9] J. Cong, M. Xie and Y. Zhang, "An Enhanced Multilevel Routing System," *Proc. IEEE International Conference on Computer Aided Design*, San Jose, California, pp. 51-58, November 2002

[10] Amit Singh , Malgorzata Marek-Sadowska, Efficient circuit clustering for area and power reduction in FPGAs, Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, February 24-26, 2002, Monterey, California, USA.

[11] http://cs.anu.edu.au/~bdm/nauty/

[12] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, ‘Multilevel Hypergraph Partitioning: Application in VLSI domain’, Proc. ACM/IEEE DAC, June 1997.

[13] Pongstorn Maidee, Cristinel Ababei, and Kia Bazarga, "Timing-driven Partitioning-based Placement for Island Style FPGAs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, Vol. 24, No. 3, pp. 395 - 406, Mar. 2005.

[14] S. Ghiasi , E. Bozorgzadeh , S. Choudhuri , M. Sarrafzadeh, A unified theory of timing budget management, Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, p.653-659, November 07-11, 2004

[15] E. Bozorgzadeh , S. Ghiasi , A. Takahashi , M. Sarrafzadeh, Optimal integer delay budgeting on directed acyclic graphs, Proceedings of the 40th conference on Design automation, June 02-06, 2003, Anaheim, CA, USA.

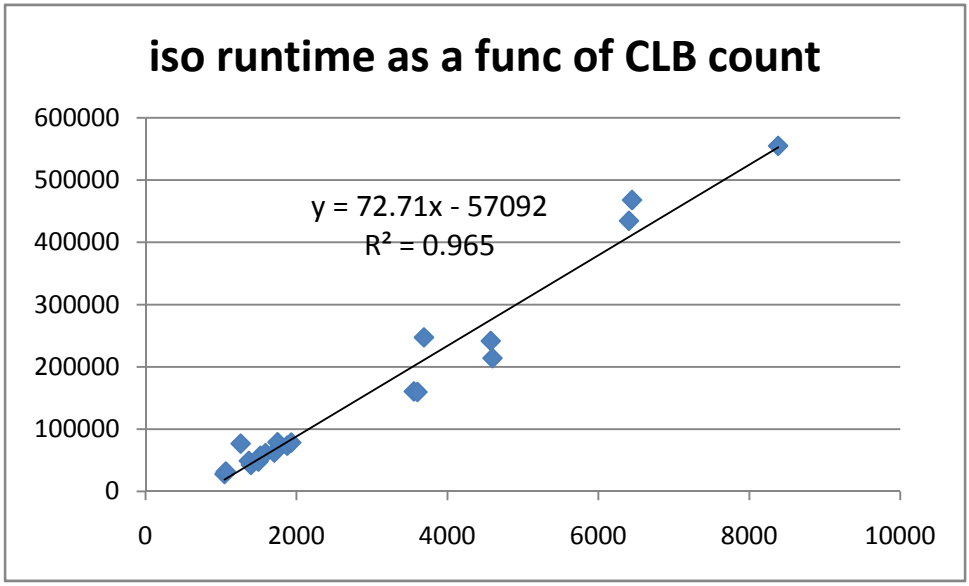

Figure 5 Runtime trend of our algorithm