Identifying and Placing Heterogeneously-sized Cluster Groupings Based on FPGA Placement Data

Farnaz Gharibian and Lesley Shannon
School of Engineering Science
Simon Fraser University
Email: fga7,lshannon@sfu.ca

Peter Jamieson
Department of Electrical and Computer Engineering
Miami University
Email: jamiespa@miamioh.edu

***Abstract*—Field Programmable Gate Arrays (FPGAs) CAD flow run-time has increased due to the rapid growth in size of designs and FPGAs. Researchers are trying to find new ways to improve compilation time without degrading design performance. In this paper, we present a novel approach that identifies tightly grouped FPGA logic blocks and then uses this information during circuit placement. Our approach is an orthogonal optimization applicable in incremental design and physical optimization, and reduces placement run-time. Specifically, we present a new algorithm that analyzes designs post-placement to extract medium-grained super-clusters that consist of two to seventeen clusters, which we call "*gems*". We modified VPR's simulated annealing placement algorithm to place our mixture of gems and clusters. Our new "Singularity Annealing" algorithm first crushes each cluster grouping into a "singularity" (treated as a single cluster). Then, the Singularity Annealer is run over this condensed circuit to obtain an initial placement, followed by an expansion of the singularities. Finally, we run a second low-temperature annealing phase on the entire expanded circuit. Our results show that our system reduces placement run-time on average by 17% while maintains the designs critical path delay, and increases designs channel width, and wirelength by 2% and 6.3%, respectively. We have also presented a test case to show the re-usability of gems in an incremental design example.**

I. Introduction

Placement is one of the most time consuming steps in the FPGA CAD flow [1] and finding an optimal solution for the FPGA placement problem is an NP-hard problem [2]. Many studies have tried to improve placement algorithms over the past 30 years, but current algorithms are far from optimal [3]. Due to the rapid growth in the size of designs and FPGAs, compilation times approach a day for large devices, significantly limiting designer productivity [4].

Several previous studies have proposed a hierarchical placement approach, in which a design is divided into coarse-grained sub-components that can be independently placed. Such an approach may reduce run-time and may scale well on multicore architectures. A primary challenge with hierarchical placement approaches, however, is identifying appropriate sub-components. At one extreme, work such as [5] assumes very small fixed-sized components, which may or may not match the locality present in circuits. At the other extreme, other work such as [6] has proposed using large IP blocks as sub-circuits during the hierarchical placement process; for circuits containing few (or no) large IP blocks, this approach may limit the amount of parallelism possible.

In this paper, we propose a technique that falls between these two extremes. We demonstrate a new approach that generates *heterogeneous* cluster groupings (which we call *gems*) based on an analysis of post-placement proximity data that finds low-level relationships between a circuit's logic blocks without enforcing a complete partitioning of the entire design or using floorplanning. In our experiment, each gem contains between two and seventeen logic clusters based on parameters for gem extraction. Compared to work such as [5], these medium-sized gems are expected to lead to better quality results, since the nature of the gems is circuit-specific. At the same time, these gems are significantly smaller than the IP blocks considered in [6]; we anticipate that for many circuits, this will lead to better parallelism.

Our design flow is as follows. The design is first compiled as normal, creating an initial placement. While the designer debugs the existing design/creates a new design module, our "gem" detection algorithm then finds cluster groupings of closely co-related clusters based on the existing placement *offline*. In a typical design project, the user performs many compilation iterations. Between each iteration, the designer refines the design (often as a result of debugging), however, much of the circuit remains unchanged. If appropriate high-quality cluster groupings from an existing design can be identified offline after a synthesis phase (i.e. when the designer is debugging the current circuit/designing new components), they can be re-used in subsequent iterations to reduce compilation time and improve the overall design productivity.

In subsequent iterations, these pre-determined gems are placed as a single unit by our novel *Singularity Placer*, reducing the size of the placement problem (and hence the run-time) for these subsequent iterations. As such, the time required for the first compilation remains unchanged since gem detection is performed offline in the background after this synthesis phase is completed and while the designer is completing the next design iteration. However, subsequent compile iterations for the same project can be accelerated by, on average, more than 17%, while maintaining the same critical path and increasing the channel width by only 2% and wirelength by only 6.3% , even though it is as if the circuit had been recompiled from scratch each time.

The specific contributions of this paper are:

- An algorithm that obtains inter-cluster proximity data

from post-placement analysis to identify highly co-related clusters ("gems") that negatively impact placement results if their clusters are not co-located.
- Our Singularity Placement (SP) algorithm, a two-stage placement algorithm that uses gems to accelerate placement. Specifically, we alter VPR's simulated annealer to treat the gems as a single cluster (singularity) during placement. In the second phase, the gems are re-expanded to their actual size for a low temperature second annealing phase over the entire circuit.
- A demonstration of how the gem identification algorithm and the Singularity Placement algorithm can be integrated into an FPGA CAD flow to improve overall design productivity by more than 17% on average.
- An example experiment to show how our gems can be used in incremental design.

This paper is organized as follows. Section II discusses previous work in placement and Section III presents our technique to extract gems based on a set of parameters. Section IV describes our singularity placer. Our experimental framework, including CAD flow and benchmarks, and the results of our analysis are in Section V. Finally, Section VI concludes the paper and summarizes future work.

II. Background

In a study done by Chang et. al. [7], the optimality and scalability of three placers from academia (Dragon [8], Capo [9], and mPL [10]), and one industrial placer (QPlace [11] from Cadence) were investigated. The results from these placers were compared for placement benchmarks that were constructed with known upper bound on optimal wirelength [12]. Their results show that existing placements solutions are not stable because their effectiveness depends on the characteristics of the benchmarks [13]. Complementary work from Cong et. al. [3] suggested that hierarchical/multilevel methods need to be used to increase the scalability of placement algorithms to catch up with the rapid capacity growth in the size of circuit designs and FPGAs.

To have a high quality multilevel placement, it is important to be able to intelligently subdivide the design into appropriate subcomponents. In particular, it is extremely valuable to know which clusters within a design should be grouped together or placed near each other to ensure good design performance. The clusters that are grouped together should have a close relation with each other and be closely co-located within in the global circuit placement. Unfortunately, it is difficult to visualize how the locations of clusters are related within the final placement because of the growth in FPGA capacity and circuit design complexity. In this work, we find an algorithm to extract these closely related clusters in circuits.

Recently, different approaches have been studied to improve run-time and quality of FPGA placement. We have categorized these approaches into the following three groups.

Partitioning/Clustering methods

The Ultra-Fast Placement algorithm aims to improve the run-time of VPR's Simulated Annealing (SA) [14] placer by initially performing multi-level clustering [5]. In Ultra-Fast, the coarse-grain block sizes at each level are *fixed* to facilitate the exchange of clusters during the swapping moves in SA. In our previous work [15], message-passing clustering [16] was used to create coarse-grain blocks. The coarse-grain blocks are grown based on a connectivity-based scoring function determined by two components: the connection between the blocks and the number of nets that are absorbed if a block is merged into the coarse-grain block. Each level is performed in two phases: a phase to build a good initial placement followed by a low temperature SA phase. We did not assume a fixed number of clusters per coarse-grain blocks (like the Ultra-Fast Placement algorithm [5]) or a fixed number of coarse-grain blocks (like the K-clustering algorithms [17]).

Applying High-level Information

SA based FPGA placement uses random initial placements. These algorithms do not consider information embedded in the original design, since circuits are typically flattened before SA is run. Some previous work in ASIC placers suggest that high-level information and design hierarchy should be considered during both clustering [18] and placement [19]. Floorplanning (or hierarchical) approaches to placement, based on the design's hierarchy as specified in its RTL have been introduced [20] and [21]. Cong [22] suggests that RTL floorplan for FPGAs may not work very well; however, the quantification of the theory is not provided. In our previous work [15], the possibility of finding coarse-grain blocks in the final placement and relating them to high-level structures such as *Module*, *Always*, and *If* was investigated. Our results showed that high-level structures are not a good candidate to partition the design and there is a close relationship between circuit design and the relations of HDL structures.

Parallel approaches

Parallelization methods are another approach to accelerate FPGA CAD flow including placement. The objective of this approach is to ensure that the implementation is scalable to leverage the number of cores available on modern parallel machines, while ensuring that the final results are deterministic independent of the number of processors. There are various parallel approaches based on simulated annealing including [1] and [23]. The work presented by Wang et al. [24] achieved a speedup of 123x when using 25 threads compared to VPR 5.0 [25] with quality degradation of 8% increase in the critical path delay and a 11% increase in the bounding box metrics. The work presented in [26] improved the timing-driven parallel placement algorithm described by [24] by reaching the best speedup by using only 16 threads instead of 25 threads.

III. How to extract GEMs?

As stated previously, coarse-grain cluster groupings (e.g. representing large IP blocks) can be identified from design files and placed using floor planning algorithms. However, we hypothesize that medium-grain cluster groupings that do not encompass the entire design exist, but at present there is no method for finding these structures. The first step to

finding high-quality medium-grain groupings is to find the *gems* in a design: tightly co-located clusters that consistently occur in placement solutions, impacting the design's critical path and/or wirelength when they are ignored. Detecting and understanding characteristics of these "gems" will help us to partition designs into coarser grain groupings. We use Manhattan distance as the proximity measure between clusters where an FPGA cluster is a group of Basic Logic Elements (BLEs) [25]. Each cluster has a unique location on FPGA after placement and the Manhattan distance between $cluster(i)$ and $cluster(j)$ location on the FPGA is calculated using Equation 1:

$$|X_{cluster(i)} - X_{cluster(j)}| + |Y_{cluster(i)} - Y_{cluster(j)}| \quad (1)$$

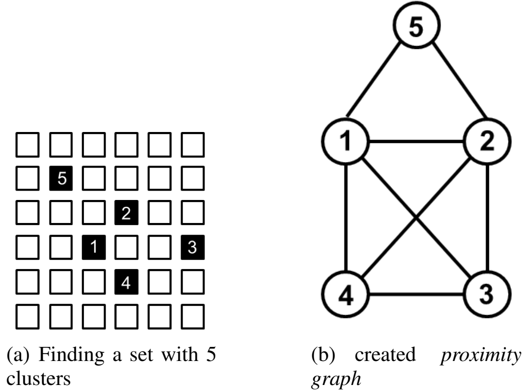
Gems are collections of clusters that are closely co-located and we find these by analyzing proximity in repetitive placements and analyzing clusters based on the following parameters:

- *num-run*, total number of placement runs which are used to create gems,
- *minimal-distance*, threshold manhattan distance to consider a pair of clusters as part of a gem, and
- *recurrence*, the percentage of the placement solutions that satisfy a minimal-distance condition.

A. Creating the Proximity Graph

We use a placed design and extract a "proximity graph", which we define as an undirected graph where each cluster is a node in the graph. Each node in the "proximity graph" is connected if the Manhattan distance of a pair of clusters is less than or equal to *minimal-distance* parameter. For example, selecting a *minimal-distance* parameter of 3 means that for a candidate cluster set, each pair of clusters in the set has a Manhattan distance less than or equal to 3. Doing this for all clusters creates the "proximity graph".

Figure 1 shows an example of five clusters placed on FPGA and their created proximity graph for *minimal-distance* parameter of 3. In Figure 1(a), all five clusters are not fully connected with *minimal-distance* of 3 because the Manhattan distance between cluster 3 and cluster 5 is 6, and Manhattan distance between cluster 4 and cluster 5 is 5, which does not satisfy the *minimal-distance* condition. In this example, clusters 1, 2, 3, and 4 are considered a connected set, and similarly, clusters 5, 1, and 2 are another. Figure 1(b) shows *proximity graph* representing clusters in Figure 1(a).



(a) Finding a set with 5 clusters (b) created *proximity graph*

Fig. 1. Example of a candidate set with *minimal-distance = 3* and its created *proximity graph*

B. MAX-CLIQUE Algorithm [27] for Extracting Gems

Finding gems is the process of finding fully connected graphs in the *proximity graph* created from our placement files with a specified size (*minimal-distance*). To find these gems, we use an algorithm called Max-Clique, which finds fully connected sub-graphs. A clique is a set of nodes in which all nodes are connected to each other. A clique *clique1* is called a maximal clique if there is no other clique in the graph that is a super set of *clique1*.

Inputs of the algorithm are, R, a set which contains a clique, P, the set of all the potential nodes not yet in a clique that should be investigated, and X, the set of all processed nodes that are included in saved maximal cliques. At the beginning, P contains all the nodes in the graph, and R and X are empty sets. Note that X is used to ensure that the algorithm only finds the maximal cliques. After the first call of the function $Clique$, the set P is reduced to only those nodes that are connected to all the nodes in R.

The algorithm recursively calls itself for the reduced set of P. When P is empty, the algorithm has found a clique in R. In that stage, if X is empty it means that R is a maximal clique, and therefore, R is saved as a gem. In the case that P is not empty, for each *element* in P, the new sets of P_new, X_new, and R_new are created and the function calls itself with these new parameters. R_new contains all nodes in R plus the *element*. P_new contains all nodes in P minus $exclude(P, element)$. The function $exclude(P, element)$ is a set of all nodes in P that are not connected to the *element*. X_new contains all nodes in X minus $exclude(P, element)$. After the recursion call, the *element* is added to X because the maximal clique is found for it and also *element* is removed from P since the investigation is done for the *element*.

IV. Singularity Placement

After gem extraction is completed for each benchmark, the extracted gems are used by our singularity annealing placement. The Singularity Placer (SP) initially maps a mixture of clusters and gems to physical cluster locations on the FPGA, without trying to bin or map the gems to specific areas of the FPGA; instead the gems are expanded during a second placement phase after which the final placement is legalized. This is unlike the previously proposed incremental Simulated Annealing algorithms [28], [29], which move clusters to a bin structure during the refinement levels.

The pseudo code for Singularity Placement is shown in Algorithm 1. In detail, the algorithm starts by crushing all the clusters within a gem to a single cluster called a singularity cluster (Line 2). Next, the placer places all singularities and clusters on the FPGA (Line 3). We have used a modified version of VPR's SA algorithm to place singularity clusters and regular clusters (Line 4). The key modification for the SA algorithm is to provide the annealer with an estimation of the wirelength and criticality for a singularity cluster. At present, we use the same model for criticality measurements as VPR

Algorithm 1 Singularity Placement

1: **function** SINGULARITY PLACEMENT(Circuit Netlist, Extracted Gems)
2: Create Singularity Clusters
3: Initial Random Placement of Singularity and Regular Clusters
4: Run Singularity SA
5: Expand Singularity Clusters
6: **if** Exceeding Column **then**
7: Shrink COLS
8: **end if**
9: **if** Exceeding Row **then**
10: Shrink ROWS
11: **end if**
12: Run Low Temperature SA
13: **end function**

except that we calculate the sum of the criticality of all the clusters in a gem when a gem is being swapped. This is only an approximation of the gem's criticality.

After placing this modified circuit, an expansion phase is used to assign all of the blocks of the circuit to a real physical location on the FPGA (Line 5). To achieve this, a larger intermediate FPGA is used. Each singularity cluster is expanded in a square-like shape. The location of a singularity cluster in the intermediate FPGA is adjusted based on other singularity clusters to avoid any location overlap between expanding singularity clusters. Therefore, the location of some blocks may be bigger than an FPGA made to fit the circuit. When all singularity clusters are expanded with no overlap and all other blocks are located in empty spaces based on their original location, the algorithm shrinks the intermediate FPGA to fit all the blocks in the original FPGA (Lines 6 through 11). Finally, a temperature SA phase, with fewer iterations is used to refine the placement of the expanded singularities with respect to clusters inside and outside the singularity (Line 12).

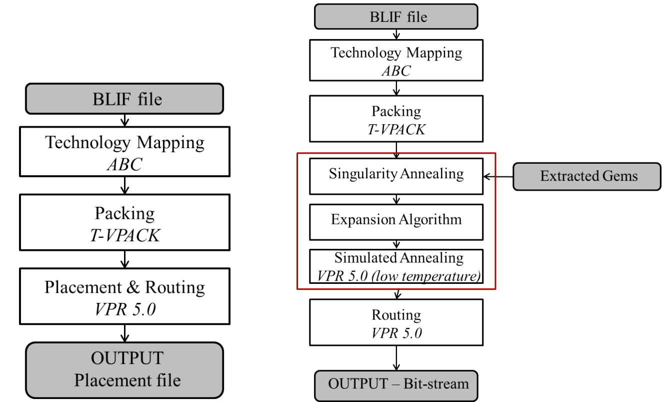
V. RESULTS

This section describes our experimental methodology and setup and then discusses the results.

Figure 2(a) shows the traditional FPGA CAD flow that is used for extracting gems. Each benchmark is passed into ABC [30] for logic optimization and mapping to Look-Up Tables (LUTs). Next, T-VPack [31] packs the LUTs and registers into clusters. The output of T-VPack is then placed and routed using VPR 5.0 [25]. The FPGA architecture that we use in these experiments consists of clusters containing ten (N = 10) 4-input LUTs (K = 4) and the routing is uni-directional. We have also used the default routing parameters given with the VPR 5.0 release, specifically: F_{cout} = 0.1, F_{cin} = 0.15, and F_s = 3.

A. Experimental Methodology

Similarly, Figure 2(b) shows the FPGA CAD flow used in our singularity placement experiments. The same tools are used to map the benchmarks for placement, and the same FPGA parameters are used as described above. The key difference between this flow and traditional flow (shown in Figure 2(a)) is the placement step as described in Section IV. A gem file is included as part of the singularity annealer for placement which contains information about the gems.



(a) Traditional FPGA CAD flow (b) FPGA CAD flow used in Singularity Placement

Fig. 2. FPGA CAD flows used in this work

To detect the critical path delay for each benchmark in both flows, the size of the FPGA fabric and the routing channel width (W) are, respectively, set to be 120% of the minimum FPGA size and 120% of the minimum channel width required to route each of the benchmark [30]. We also incorporated fixed I/O location into the placement phase for all runs in both CAD flows.

Our benchmarks consist of 17 benchmarks, of which 8 of them are synthetic benchmarks and the rest are open source benchmarks. The synthetic benchmarks (*synth_1* through *synth_8*) are generated using a tool published by Mark et al. [32]. The open source benchmarks that are used in this experiment are distributed with Odin-II [33]. For both the traditional flow and the proposed flow, we ran each of the benchmarks through the placement algorithms ten times with ten different random seeds to find the average critical path delay. All experiments are run on servers with 8 core Intel Xeon CPU (3.16GHz) with dedicated cache and run-time results are measured in seconds from the time the placement is started to the point where placement is finished.

B. Benchmark Statistics

Table I provides more information about benchmarks and gems using following gem parameters: *recurrence = 80%*, *num-run = 20*, and *minimal-distance=6*. Column 1 contains the names of the benchmarks, and Columns 2 through 4 list the number of Clusters, the number of IOs, and number of gems for each benchmark. Columns 5 shows the percentage of clusters that are covered by gems in each benchmark. We have not considered IO blocks to create gems.

Our algorithm can detect gems even for a small *minimal-distance* value, confirming that the placement proximity data contains tightly connected clusters in the benchmarks that are repeatedly placed close together. By varying the parameters for creating *proximity graph*, we can alter the size and "quality" of the gems that are created for use in the placer. In this paper, the number of clusters in gems varies between two and seventeen

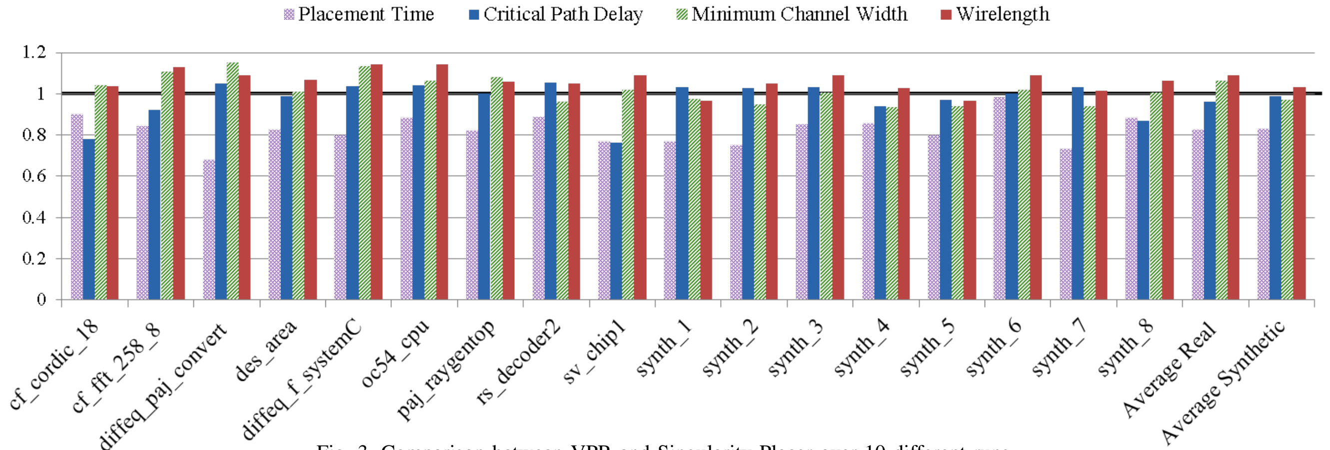


Fig. 3. Comparison between VPR and Singularity Placer over 10 different runs

TABLE I
BENCHMARKS USING FOLLOWING PARAMETERS: *recurrence = 80%*, *num-run = 20*, AND *minimal-distance=6*

Benchmark	Number of			% of covered in Gems
	Clusters	IOs	Gems	
cf_cordic_18	566	111	105	76
cf_fft_258_8	954	69	190	88
diffeq_paj_convert	396	258	47	34
des_area	229	190	26	16
diffeq_f_systemC	393	162	55	57
oc54_cpu	376	139	56	62
paj_raygentop	902	544	149	36
rs_decoder2	370	32	61	89
sv_chip1	3940	213	765	87
synth_1	2140	930	407	53
synth_2	4959	1398	958	66
synth_3	2637	516	518	75
synth_4	1136	579	230	46
synth_5	3939	1445	819	57
synth_6	1335	1149	209	10
synth_7	2161	785	352	60
synth_8	2290	1535	385	29

because of the parameters that we have considered in gem extraction.

C. Evaluation of our Singularity Placer

Here we demonstrate that by using a set of gems, the singularity placer will generate solutions with improved run-times compared to fine-grain annealing without measurably impacting the critical path delay on average.

This paper is focusing on small to medium-size cluster groupings as the percentage of them that are likely to persist between design iterations is higher. Since there are fewer clusters per gem and a relatively large number of clusters, as opposed to a few very large clusters, any small changes in design logic should have a reduced impact on the persistence of the majority of gems. For this experiment, gems extracted based on the parameters listed in the previous section.

Figures 3 shows the experimental results comparing our algorithm for placing gems and VPR using real (open source) and synthetic benchmarks. The FPGA size in both experiments (VPR and our algorithm) is the same. The x-axis lists the benchmarks and the y-axis shows the normalized results of the singularity placer compare to VPR for the placement run-time, critical path delay, minimum channel width and wirelength. When the value is less than one, it indicates that singularity placer has a better performance than VPR. The las two column groups (*Average Real* and *Average Synthetic*) show average placement run-time, critical path delay, minimum channel width and wirelength for real benchmarks and synthetic benchmarks.

The results show that on average our placement run-time has been decreased by more than 17% compared to VPR over the seventeen benchmarks, while roughly maintaining the quality of placement for the critical path delay on average. Since this was the objective of the work, it suggests that further experimentation with parameter choices to determine more optimal values may not only further reduce run time while maintaining the critical path delay, but may possibly reduce the critical path delay. For example, we can better investigate the effects of varying the parameters for gem clustering; we can also investigate the scheduling and expansion steps of the singularity placer more thoroughly. Another parameters that we will need to investigate for both the traditional and proposed new flows is the possibility of reducing the SA inner loop number. Currently, both our singularity SA and VPR SA use the same inner loop number. It should be possible to reduce both values slightly without impacting the quality of the final placement; however, a thorough investigation is left to future work. Over the seventeen benchmarks, the minimum channel width increased slightly to 2% and the wirelength to 6.3%. We expect that this is the cost of maintaining the critical path delay for our proposed approach as the singularity placer will tend to result in slightly more compact designs than the traditional VPR flow- leading to greater congestion.

D. A Test case for incremental design

We ran a benchmark through our CAD flow and created gems with following parameters: *recurrence=80%*, *num-run=20*, and *minimal-distance=6*. We then added another module to the design by inserting flip flop between output of the first design and input of the added module in netlist. We reran the new design through our singularity CAD flow using the gems created from the original design. The results, averaged over 10 different runs, demonstrated a 30% decrease in placement run-time while maintaining the critical path delay and increasing minimum channel width and wirelength by 19% and 13%, respectively.

VI. Conclusion and Future Work

In this paper, gems are defined as closely related clusters in designs with two characteristics: 1) the clusters in a gem are co-located with a small Manhattan distance separating them from each other, and 2) gems cover a portion of the circuit, but their relationship is the key to placement quality (critical path and/or wirelength). We have used different parameters: *minimal-distance*, *num-run*, and *recurrence* to create a *proximity graph*, that then can be used to find gems using a Max-Clique algorithm.

We illustrate that post-placement proximity data can be evaluated to create gems, similar to partitioning and clustering algorithms that analyze netlists. However, not only does our approach analyze post-placement data, but it generates heterogeneous cluster groupings, without having to partition the entire design. This non-aggressive approach to clustering then allows the placer to swap larger groupings. We demonstrated a novel two phase placer, called the singularity annealer for using gems to improve placement run time by an average of 17.7% for our sample benchmarks and 17.3% for our synthetic benchmarks with minimally improved critical path delays on average. On average, the minimum channel width increased by 6.3% for the sample benchmarks and decreased by 2.8% for the synthetic benchmarks, for an overall average increase of 2%. Wirelength has increased on average by 9.0% on real benchmarks and 3.3% for synthetic benchmarks, for an overall average increase of 6.3%. These results suggest that we may achieve more significant gains in run time and/or circuit quality (i.e. critical path delay/channel width) after future investigations into the tradeoffs of various parameter settings generating gems and tuning the singularity placer.

There are a numerous opportunities for future work. In particular, we are interested in the following areas. First, can we apply our approach to clustering on netlist data to reduce compilation time while maintaining quality. Second, how does our approach to clustering compare to global clustering/partitioning algorithms such as iRAC [34] and hMETIS [35]. Finally, we wish to further investigate the creation of gems and their affects on placement by varying different parameters such as the size of gems, the inner-num values of SA.

References

[1] A.Ludwin, V.Betz *et al.*, "High-quality, deterministic parallel placement for fpgas on commodity hardware," in *Proceedings of the 16th International ACM/SIGDA Symposium on FPGAs*, 2008, pp. 14–23.

[2] V.Betz, J.Rose *et al.*, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[3] J.Cong, J. R.Shinnerl *et al.*, "Large-scale circuit placement," *ACM Transactions on Design Automation of Electronic Systems*, vol. 10, no. 2, pp. 1–42, 2005.

[4] S.Chin and S.Wilton, "Towards scalable fpga cad through architecture," in *ACM Int'l Symp on FPGAs*, 2011, pp. 143–152.

[5] Y.Sankar and J.Rose, "Trading quality for compile time: ultra-fast placement for fpgas," in *Int'l Symp on FPGAs*, 1999, pp. 157–166.

[6] C.Lavin, M.Padilla *et al.*, "Using hard macros to reduce fpga compilation time," in *on the Proc. of the FPL*, 2010, pp. 438–441.

[7] C. C.Chang, J.Cong *et al.*, "Optimality and scalability study of existing placement algorithms," in *In Proc of the ASP-DAC*, 2003, pp. 621–627.

[8] M.Wang, X.Yang *et al.*, "Dragon2000: Standard-cell placement tool for large industry circuits," in *In Proc. of the IEEE Int'l Conf on CAD*, 2000, pp. 260–264.

[9] A. E.Caldwell, A. B.Kahng *et al.*, "Can recursive bisection produce routable placements?" in *In Proc of DAC*, 2000, pp. 477–482.

[10] T. F.Chan, J.Cong *et al.*, "An enhanced multilevel algorithm for circuit placement," in *In Proc of the IEEE Int'l Conf on Computer-Aided Design*, 2003.

[11] C. D. S.Inc, "Qplace version 5.1.55, compiled on 10/25/1999," in *Envisia ultra placer reference*, 1999.

[12] J.Cong, M.Romesis *et al.*, "Optimality and stability of timing-driven placement algorithms," in *In Proceedings of the IEEE International Conference on Computer Aided Design*, 2003.

[13] ——, "Optimality, scalability and stability study of partitioning and placement algorithms," in *In Proceedings of the International Symposium on Physical Design*, 2003, pp. 88–94.

[14] S.Kirkpatrick, C. D.Gelatt *et al.*, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.

[15] F.Gharibian, L.Shannon *et al.*, "Analyzing system-level informations correlation to fpga placement," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 6, no. 3, October 2013.

[16] B. J.Frey and D.Dueck, "Clustering by passing messages between data points,," in *Science*, vol. 315, Feb. 2007, pp. 972–976.

[17] J. B.MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1. University of California Press, 1967, pp. 281–297.

[18] D.Behrens, K.Harbich *et al.*, "Circuit partitioning using high-level design information," in *Conference on Integrated Design - process Technology*, 1996, pp. 259–266.

[19] Y. W.Tsay, W. J.Fang *et al.*, "Preserving hdl synthesis hierarchy for cell placement," *Proceedings of the 1997 International Symposium on Physical design*, pp. 169–174, 1997.

[20] J. M.Emmert and D.Bhatia, "A methodology for fast fpga floorplanning," in *FPGA 1999*.

[21] R.Tessier, "Fast placement approaches for fpgas," *ACM Trans. on Design Automation of Electronic Systems*, vol. 7, no. 2, pp. 284–305, April 2002.

[22] J.Cong, "Timing closure based on physical hierarchy," in *Proceedings of the International Symposium on Physical Design*, 2002, pp. 170–174.

[23] M.Haldar, A.Nayak *et al.*, "Parallel algorithms for fpga placement," in *10th Great Lakes Symposium on VLSI*, 2000, pp. 86–94.

[24] C.Wang and G.Lemieux, "Scalable and deterministic timing-driven parallel placement for fpgas," in *In Proc of the 19th ACM/SIGDA Int'l Symp on FPGAs*, 2011, pp. 153–162.

[25] J.Luu, I.Kuon *et al.*, "VPR 5.0: FPGA CAD and Architecture xploration Tools with Single-Driver Routing, Heterogeneity and Process Scaling," in *ACM/SIGDA International Symposium on FPGAs*, Feb 2009.

[26] J.Goeders, G.Lemieux *et al.*, "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition," in *ReConFig*, 2011, pp. 41–48.

[27] C.Bron and J.Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *ommunications of the ACM, ACM Press: New York, USA.*, vol. 16, no. 9, pp. 575–577, 1973.

[28] C.c.Chang, I.Cong *et al.*, "Multi-level placement for large-scale mixed-size ic designs," in *ASPDAC 2003*, 2003, pp. 325–330.

[29] C.-C.Chang, J.Cong *et al.*, "Physical hierarchy generation with routing congestion control," in *In ACM/SIGDA International Symposium on Physical Design (ISPD)*, 2002, pp. 36–41.

[30] A.Mishchenko, S.Chatterjee *et al.*, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Transactions on CAD*, vol. 26, no. 2, pp. 240–253, 2007.

[31] A.Marquardt, V.Betz *et al.*, "Using Cluster-Based Logic Blocks and Timing-Driven Packing to Improve FPGA Speed and Density," in *ACM/SIGDA Int'l Symp on FPGAs*, Monterey, CA, 1999, pp. 37–46.

[32] C.Mark, A.Shui *et al.*, "A system-level stochastic circuit generator for fpga architecture evaluation," in *FPT 2008*, pp. 25–32.

[33] P. A.Jamieson, K. B.Kent *et al.*, "Odin II - An Open-source Verilog HDL Synthesis Tool for Academic CAD Flows," in *IEEE Symp on FCCM*, 2010.

[34] A.Singh and M.Marek-Sadowska, "Efficient Circuit Clustering for Area and Power Reduction in FPGAs," in *FPGA 2002*, pp. 59–66.

[35] G.Karypis, R.Aggarwal *et al.*, "Multilevel Hypergraph Partitioning: Application in VLSI Domain," in *DAC '97: Proceedings of the 34th ACM/IEEE conference on Design automation*, 1997, pp. 526–529.