Scalable Flip-Flop Clustering Using Divide and Conquer For Capacitated *K*-Means

Andrew B. Kahng
University of California, San Diego
La Jolla, California, USA
abk@ucsd.edu

Sayak Kundu
University of California, San Diego
La Jolla, California, USA
sakundu@ucsd.edu

Shreyas Thumathy
Precision Innovations Inc.
San Diego, California, USA
shreyasthumathy@gmail.com

ABSTRACT

Multi-bit flip-flop clustering is a well-studied optimization problem in physical design: carefully merging multiple single-bit flip-flops into a single multi-bit flip-flop can decrease the total power consumption in a clock distribution network due to lower clock power and routed clock wirelength. We propose a pointset decomposition heuristic that in conjunction with capacitated k-means [4] enables a scalable, divide-and-conquer flow for multi-bit flip-flop clustering. Our flow produces high-quality flip-flop clustering and placement solutions with respect to total power consumption, area, timing, and wirelength metrics evaluated after the post-routing optimization (PRO) stage of P&R. We test our flow on five designs of varying input size (0.5K to 64K clusterable single-bit flip-flops) implemented using the ASAP7 7nm research enablement [3] [9]. Empirical results show that our new flow is competitive with current state-of-the-art flows. Compared to MeanShift [2], we achieve 6.18% (resp. 1.90%) maximum (resp. average) reduction in total power consumption, along with improved *total negative slack* and wirelength. Compared to FlopTray [4], we achieve a 400× speedup on larger designs such as VGA (17K single-bit flip-flops), but with an average 1.12% degradation in total power consumption.

ACM Reference Format:
Andrew B. Kahng, Sayak Kundu, and Shreyas Thumathy. 2024. Scalable Flip-Flop Clustering Using Divide and Conquer For Capacitated *K*-Means. In *Great Lakes Symposium on VLSI 2024 (GLSVLSI '24), June 12–14, 2024, Clearwater, FL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3649476.3658744

1 INTRODUCTION

Multi-bit flip-flop (MBFF) clustering is a technique that reduces power consumption in a clock distribution network. Each *single-bit flip-flop* (FF) is mapped to a single *slot* in an MBFF, which results in lower power consumption due to the reduced number of clock sink pins and routed clock wirelength.[1] For example, if every four single-bit FFs are clustered into a single 4-bit MBFF, then the number of clock pins in the network is reduced by 75%. Figure 1 illustrates the clustering of two FFs into a 2-bit MBFF. Current state-of-the-art works include FlopTray (FTray) [4] and MeanShift (MShift) [2]. FTray achieves high quality of results (QoR), but has long runtimes from solving large min-cost flow instances. MShift achieves clustering solutions very quickly but is oblivious to MBFF slot locations; this potentially incurs power and timing overheads. Thus, in our present work we seek a new method that can achieve high QoR while also maintaining reasonable runtimes. Notably, we develop a hierarchical pointset decomposition heuristic that creates much smaller min-cost flow instances than in [4] without undue loss of solution quality. Our contributions include the following.

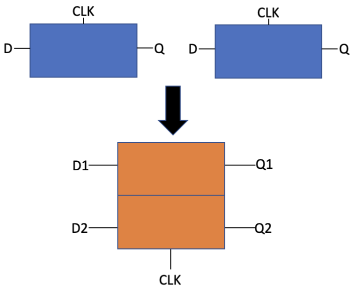


Figure 1: Two single-bit flip-flops are clustered into a 2×1 **multi-bit flip-flop.**

- We propose a new pointset decomposition heuristic that enables a scalable divide-and-conquer approach to MBFF clustering. In particular, we enable very substantial speedups of min-cost flow in the FTray method of [4].
- We recalibrate the α and β parameters of FTray's integer linear program (ILP) setup, to fit the ASAP7 7nm research enablement. These parameters govern clustering and placement solutions, so recalibration is needed to properly assess our methods.
- Extensive experimental evaluations using the ASAP7 enablement show that our methods achieve high post-P&R quality of results, as measured using *Cadence Innovus v21.1.*
- Our flow achieves up to 6.18%, and 1.90% on average, improvement in total power consumption compared to MShift when using up to 16-bit MBFFs. Timing (total negative slack) also improves on average compared to MShift.
- Compared to FTray, our flow achieves an average speedup of 100×, and on larger designs such as VGA, it is up to 400× faster.

In the following, Section 2 reviews recent state-of-the-art MBFF clustering works. Section 3 formulates our problem, and Section 4 describes our proposed method. Sections 5 and 6 respectively present our experimental design and experimental results. We conclude in Section 7.

2 RELATED WORK

Several works have addressed the problem of effective MBFF clustering at the post-placement stage.

[1] When a single-bit FF is mapped to a slot in an MBFF, the FF is removed from the netlist and replaced with the MBFF slot.

GLSVLSI '24, June 12–14, 2024, Clearwater, FL, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0605-9/24/06
https://doi.org/10.1145/3649476.3658744

[4] proposes a capacitated k-means flow (FTray) which starts by finding a starting set of MBFF locations, based on multiple runs of a single iteration of k-means++ [1]. The k-means++ run with the highest Silhouette score [6] is chosen, and the selected locations are inputs for multiple iterations between minimum-cost flow (MCF, whereby each FF is assigned to an MBFF slot), and linear programming (LP, whereby each MBFF's center location is adjusted to minimize the total displacement of FFs to their assigned MBFF slots). The flow terminates by solving an integer linear program (ILP) that places MBFFs so as to minimize a weighted sum of total power usage, total displacement of FFs to their assigned MBFF slot, and total sum of relative displacements of launch-capture FFs in timing-critical paths. This work yields high QoR, but is susceptible to very long runtimes due to the massive search space from which the MCF is induced.

[2] proposes a flow (MShift) which combines mean shift and stable matching. The flow starts with the execution of *effective mean shift*, which determines fixed k-nearest neighbors (KNN) and bandwidth parameters. The stable matching component then assigns FFs to MBFFs; in the open-source implementation of MShift [18], a separate "capacity" variable governs the tradeoff between runtime and QoR.

Although MShift is scalable, the algorithm is oblivious to timing-critical path information and MBFF slot locations. On the other hand, [4] demonstrates that such information can be exploited to improve timing and total power consumption outcomes of MBFF clustering. Furthermore, with MShift large clusters can be formed when FF-to-MBFF center displacements are reasonably small, since actual slot locations are not considered; this can further spoil timing.

Last, [8] proposes a pointset decomposition-like algorithm to locate potential MBFFs using k-means. This flow also risks creating very large clusters that can potentially spoil timing [2]. And, poor clustering solutions can form as a result of k values that are influenced by a user parameter (*max-split*) rather than the intrinsic characteristics of the given pointset.

Table 1: Notations.

Notation	Definition
L	Number of clusterable FFs.
F	Set of all L clusterable FFs, $F = \{f_1, f_2, ..., f_L\}$.
X'_l, Y'_l	X and Y coordinates of f_l.
N	Number of candidate flop trays (MBFFs).
C_i	Total cost (total power consumption) of MBFF i.
S_i	Size, i.e., number of bits (slots), of MBFF i.
E_i	Boolean value indicating if MBFF i is used.
$X_{i,j}, Y_{i,j}$	X and Y coordinates of slot j in MBFF i.
$B_{l,i,j}$	Boolean value indicating if f_l is mapped to slot j in MBFF i.
$MCF(l, i, j)$	Flow on the edge from f_l to slot j in MBFF i in the minimum-cost flow solution.
M	Number of timing-critical paths (i.e., to clusterable FF endpoints).
P_m	The starting (launch) FF of the m^{th} timing-critical path.
P'_m	The ending (capture) FF of the m^{th} timing-critical path.
$d_x(l), d_y(l)$	X and Y displacements between f_l and its assigned MBFF slot.
D	Sum of absolute displacements for all F FFs to their assigned MBFF slots.
W	Total power consumption of all used MBFFs.
R	Total relative displacements of all M timing-critical paths.
U	Upper bound on pointset size (in the pointset decomposition solution) in our proposed flow.

3 PROBLEM FORMULATION

We now define the flip-flop clustering problem and review the ILP formulation of [4]. Notations are summarized in Table 1.

Problem: Given a set $F = \{f_1, f_2, ..., f_L\}$ consisting of L clusterable FFs with placed locations and timing estimates, cluster every FF into a placed MBFF of available size, e.g., from a set of sizes $\{1, 2, 4, 8, 16\}$. Previous works focus on optimizing metrics such as area, power, and timing [2][4][8].

Our objective function follows [4], i.e., we use an integer linear program (ILP) to assign FFs to MBFF slots, so as to minimize a weighted sum of (1) power usage, (2) total sum of displacements of FFs to their assigned MBFF slots, and (3) total sum of relative displacements of the launch-capture FF pairs of M timing-critical paths. Total power usage is given by Equation (1):

$$W = \sum_{i=1}^{N} E_i \cdot C_i \tag{1}$$

where E_i is a binary value indicating if the candidate MBFF i is used. In [4], the number of candidate MBFFs (N) is bounded by L. Specifically, $N = L + \lceil \frac{L}{2} \rceil + \lceil \frac{L}{4} \rceil + ...$, where each denominator corresponds to an available MBFF size (e.g., 2, 4, 8, 16). The sum of displacements that corresponds to a given assignment of FFs to MBFF slots is given by Equation (2) below.

$$D = \sum_{l=1}^{L} |d_x(l)| + |d_y(l)| \tag{2}$$

$$d_x(l) = \sum_{i=1}^{N} \sum_{j=1}^{S_i} (X_{i,j} - X'_l) \cdot B_{l,i,j} \quad \forall l \tag{3}$$

$$d_y(l) = \sum_{i=1}^{N} \sum_{j=1}^{S_i} (Y_{i,j} - Y'_l) \cdot B_{l,i,j} \quad \forall l \tag{4}$$

Equations (3) and (4) respectively give x- and y-displacements between an FF and its assigned MBFF slot. $B_{l,i,j}$ is a binary value indicating if f_l is mapped to slot j in MBFF i. Finally, the sum total of relative displacements for launch-capture FF pairs in the M timing-critical paths is given by Equation (5):

$$R = \sum_{m=1}^{M} r_x(m) + r_y(m) \tag{5}$$

$$r_x(m) = |d_x(P_m) - d_x(P'_m)| \quad \forall m \tag{6}$$

$$r_y(m) = |d_y(P_m) - d_y(P'_m)| \quad \forall m \tag{7}$$

where P_m and P'_m respectively represent the start and end FFs on the m^{th} timing-critical path. Equations (6) and (7) respectively give the relative x- and y-displacements between an FF and its assigned MBFF slot, for each given timing-critical path. The optimization objective (i.e., ILP objective function) is given by Equation (8).

$$\alpha * W + D + \beta * R \tag{8}$$

We note that [4] studied a commercial 28nm FDSOI technology, and used parameters $\alpha = \{20, 40, 60, 80\}$ and $\beta = 1$. In Section 4.2, we recalibrate α and β to match ASAP7 [3], which is a research proxy for 7nm node enablement.

In the next section, we develop an MBFF clustering algorithm that is fast, scalable, and capable of delivering strong timing and power quality of results – on par with recent state-of-the-art methods such as FTray [4] and MShift [2].

4 PROPOSED METHOD

To achieve our objective of a high quality, scalable MBFF clustering algorithm, we propose a pointset decomposition algorithm that enables a divide and conquer approach for MBFF clustering via capacitated k-means [4].

4.1 Our Flow

Pointset Decomposition. Our simple yet effective recursive pointset decomposition heuristic splits all L FFs into sub-problems containing $\leq U$ FFs. This heuristic is the main contribution of our work. Our method first runs k-means++ with all values of $k \in [2, 8]$. Following [7],we use the Silhouette metric to find a best value of $k \in [2, 8]$.[2] We run a multi-start strategy using the k with the highest Silhouette score. Then, we select the clustering solution that yields the lowest average FF to cluster center displacement.[3] Our algorithm then iteratively merges the two closest clusters together, as determined by Manhattan distance between the respective cluster centers, as long as the cluster size $\leq U$ (default: $U = 500$) constraint is not violated. Based on our preliminary studies, during the iterative merging of clusters we do not update the set of cluster centers when a new merged cluster is formed.[4] Pseudocode for our approach is given in Algorithm 1.

Algorithm 1: Recursive Pointset Decomposition

```
   Procedure: Decomp
   Input: A set of single-bit FFs, $F'$
   Output: A list of sets of single-bit FFs, $F''$
1  if $|F'| \leq U$ then
2      return $\{F'\}$;
3  end
4  Choose $k$ from $[2, 8]$ that maximizes the Silhouette score;
5  for $cur \leftarrow 1$ to $multistart$ do
6      $sol_{\text{cur}} \leftarrow \text{KMeans}(F', k, \text{seed})$;
       /* A better solution has lower FF-to-center distance */
7      if $(sol_{cur}$ is better than $sol_{best})$ || $(sol_{best}$ is empty$)$ then
8          $sol_{\text{best}} \leftarrow sol_{\text{cur}}$;
9      end
10 end
   /* Iteratively merge the closest pair of clusters if the size of the
      merged cluster does not exceed the cluster size upper bound $U$ */
11 $sol' \leftarrow MergeClusters(sol_{\text{best}}, U)$;
12 $F'' \leftarrow \{\}$;
13 for $i \leftarrow 1$ to $|sol'|$ do
14     $F'' \leftarrow F'' \cup Decomp(sol'_i)$;
15 end
16 return $F''$;
```

The algorithm first checks whether the current pointset is already of size $\leq U$ (Lines 1-3). We then choose the k value in the range [2,8] for k-means++ that yields the maximum Silhouette score (Line 4). Next, we run a total of *multistart* runs of k-means++ (default: *multistart* = 20) using the selected k value, with different seed values,[5] and then proceed with the clustering solution from these multi-start runs which has minimum sum of FF to cluster center distances (Lines 5-10). Next, the *MergeClusters* procedure builds a set of edges between clusters, with edge cost given by the distance between cluster centers; *MergeClusters* then performs Kruskal-like merging as long as the size of the resulting merged cluster is $\leq U$ (Line 11). Finally, we recurse on each resulting cluster, and return the results of each call (Lines 12-16). The algorithm returns F'', which is a list of sets of single-bit FFs. Throughout the following discussion, we refer to a single set of single-bit FFs as a *sub-problem*.

Each sub-problem resulting from the recursive decomposition (i.e., each set of single-bit FFs in F'') is processed using the method of [4]. More specifically, *on each sub-problem* we run min-cost flow (MCF) based capacitated k-means to assign FFs to MBFF slots; an LP to improve MBFF locations; and an ILP to choose instantiated MBFFs – all following [4].

Initial MBFF locations are generated with a single iteration of k-means++ on the set of FF locations in the sub-problem.

Minimum-Cost Flow. Following [4], for each distinct MBFF size we construct an instance of MCF. The MCF flow network matches each FF with a single MBFF slot, and the min-cost flow solution minimizes D in Equation (2). That is, D is the sum of weights (distances) over all edges to which the MCF solution assigns a *flow* of one. *Each* MCF solution (for a given MBFF size) is the starting point for MBFF location improvement, via the LP below. Because the MBFF locations have been changed, a new MCF-LP iteration can be performed. Our empirical studies show little or no benefit to using more than five MCF-LP iterations. Thus, results that we report use a default limit of five iterations. The *union* of MCF solutions obtained for all distinct MBFF sizes is used in forming the ILP described below.

Linear and Integer Linear Programs. As in [4], we use linear programming (LP) to relocate each MBFF center so as to minimize the sum of displacements of FFs to their assigned MBFF slots (Equation (2)). The LP is formed from the matching of FFs to assigned MBFF slots, i.e., the nonzero $B_{l,i,j}$ values in the MCF solution.

The ILP selects which MBFFs (of all sizes) are actually used, by reassigning binary values to $B_{l,i,j}$ in order to minimize Equation (8). We slightly modify the ILP setup from [4] to include relative startpoint/endpoint displacements for a critical path only when both the startpoint and endpoint are in the sub-problem.[6]

For completeness, Equations (9) through (13) provide the details of the ILP. All $B_{l,i,j}$ values are set to zero prior to ILP calls. The ILP assigns 0-1 values to $B_{l,i,j}$ variables, which are induced from the

[2] We examine $k \in [2, 8]$ since this is a natural range to consider for dividing a planar pointset, and since we will adopt a recursive decomposition approach. Our algorithm can be viewed as being in the continuum between recursive splitting of a pointset into two components ($k = 2$), and octant partitioning ($k = 8$). We also note that multiple k-means++ runs are executed in parallel without harming walltime.

[3] We find minimum average distance to be a valid evaluation criterion: we want FFs to have low distance to their assigned cluster centers, as this is likely to yield placements that result in low D.

[4] Thus, the cluster merging resembles Kruskal's minimum spanning tree algorithm, but with the addition of an upper bound U that blocks certain merges. We speculate that retaining all k original cluster centers after merging clusters together gives a helpful "single-linkage" flavor to the cluster merging even though k is small.

[5] k-means++ relies on an initial seed, and generates seed-dependent solutions.

[6] The ILP in [4] minimizes Equation (5) using information from all M timing-critical paths. By contrast, when we process a given sub-problem, we do not consider critical paths whose endpoints belong to different sub-problems. Experimental results in Section 6 below confirm that in practice, post-route timing results are not harmed by this. We attribute this to (i) well-optimized MBFF placements, (ii) the minimization of total displacement of FFs to assigned MBFF slots, (iii) consideration of all critical path startpoint-endpoint pairs in the sub-problem, and (iv) use of $U = 500$.

results of MCF as described above.

$$minimize(\alpha \cdot W + D + \beta \cdot R) \tag{9}$$

subject to the constraints (equivalent to those in [4])

$$\sum_{l=1}^{L} B_{l,i,j} \leq 1 \quad \forall i, j \tag{10}$$

$$E_i \leq \sum_{l=1}^{L} \sum_{j=1}^{S_i} B_{l,i,j} \quad \forall i \tag{11}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{s_i} B_{l,i,j} = 1 \quad \forall l \tag{12}$$

$$B_{l,i,j} \leq MCF(l, i, j) \quad \forall l, i, j \tag{13}$$

Equation (10) ensures that each slot j in MBFF i replaces at most one FF. Equation (11) ensures that MBFF i must be placed if any slot j in that tray replaces FF l. Equation (12) ensures that FF l must be mapped to one slot j in one MBFF i. Equation (13) ensures that we only consider 0-1 $B_{l,i,j}$ variables corresponding to edges with flow 1 in an MCF solution.

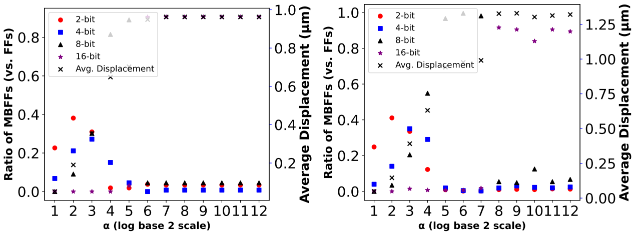
4.2 Key Implementation Details

This subsection describes several key details of our study: (i) recalibration of α and β parameters for the ASAP7 technology context; (ii) exploration and default setting of the U parameter; and (iii) experimental range used for the α parameter.

Recalibration of α and β for ASAP7. The authors of [4] determine FTray parameter values $\alpha = \{20, 40, 60, 80\}$ and $\beta = 1.0$ empirically for a commercial 28nm FDSOI enablement. Since our work uses the ASAP7 7nm research PDK and design enablement, we must recalibrate α and β for FTray to match ASAP7.

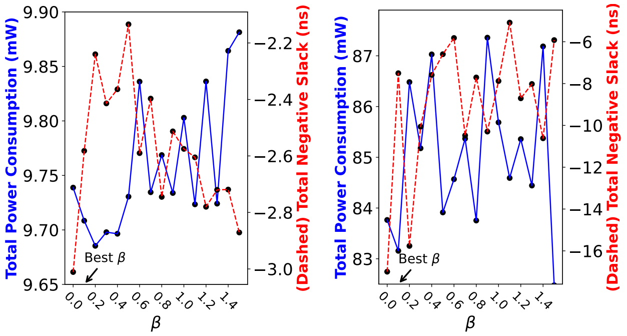
Figure 2 shows the fraction of FFs that are clustered with a particular MBFF size (2, 4, 8, 16) for the AES and JPEG designs,[7] when $\beta = 0$. (Reasonable β values are small and do not dominate the objective in Equation (8); hence, reasonable β values will lead to similar MBFF usage [4].) The figure also shows the average displacement per FF. We observe that values reported from FTray are not necessarily monotonic with respect to increasing α: (i) a relative gap is set so that the ILP solver does not spend time for minimal objective function gains; and (ii) if D or $\alpha \cdot W$ are not dominant objective function terms, then it is possible for high displacement with high MBFF usage to yield similar objective function values as when displacement and MBFF usage are both low. In our studies using ASAP7 enablement, we empirically set $\alpha = \{4, 67, 130, 193, 256\}$.

To find an appropriate β parameter value, we sweep $\beta = \{0.0, 0.1, 0.2, \ldots, 1.5\}$ with fixed $\alpha = 130$. With this α value, MBFF usage does not dominate the objective function, and FF movement has impact on the assessment. Figure 3 plots total power consumption and total negative timing slack (TNS) vs. β for the AES and JPEG designs. For every β value, we "denoise" results by running three runs with target clock period (TCP) constraint, as well as TCP±1ps; we report the median of total power and TNS values. We see that compared to the baseline (i.e., $\beta = 0$), $\beta = 0.1$ generates better results in terms of total power and TNS. In the FTray evaluation reported below, we use a default value of $\beta = 0.1$.



(a) (b)

Figure 2: FTray [4]: MBFFs instantiated (as a fraction of the original number of FFs, i.e., L) and average displacement, versus α. Designs: (a) AES, (b) JPEG.



(a) (b)

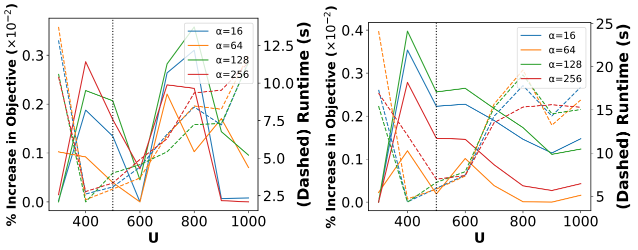
Figure 3: FTray [4]: TNS and Total Power Consumption, versus β. Designs: (a) AES, (b) JPEG.

Setting the U Parameter. In our approach, the U parameter trades off size of the search space – hence solution quality – versus runtime. We seek a default value for U that yields good clustering solutions in reasonable runtimes. Since our ILP uses Equation (8) to make FF to MBFF slot assignments, we use the ILP objective function (i.e., Equation (8)) to judge the quality of our solution.

Figure 4 shows the impact of U on QoR and runtime for the JPEG and VGA designs. We normalize QoR to the lowest observed objective function value for a given design and α. A fixed $\beta = 0$ is used; as noted above, β will not dominate the objective function, and we expect to see similar results across reasonable β values. The figure shows very similar QoR across all α values and designs, i.e., y-axis values are within a range of $\leq$ 1%. We also observe that runtimes are not necessarily monotonic in U. For large inputs, the pointset decomposition will have long runtime when U is small, due to the recursive approach and the large number of sub-problems created. Based on this and other background studies, we set a default of $U = 500$ to obtain clustering solutions that result in good QoR, with short runtimes.

Given a default setting of $U = 500$, we double-confirm the setting of α and β parameters in our method. For α, Figure 5 shows similar trends as seen for FTray, with regard to increasing average displacement and MBFF usage. However, a large α value is required in order to activate the dominance of 16-bit MBFF usage. We empirically choose to use the arithmetic progression $\alpha = \{8, 262, 516, 770, 1024\}$ for our flow, with the main purpose being to span a wide range of α values. For β, Figure 6 shows a similar effect of β on total

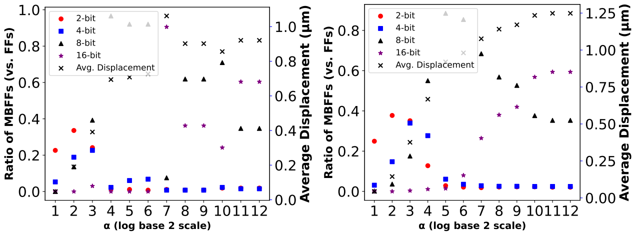
[7] This is calculated as (MBFF size) × (number of MBFF instances of that size) / L.



(a)

(b)

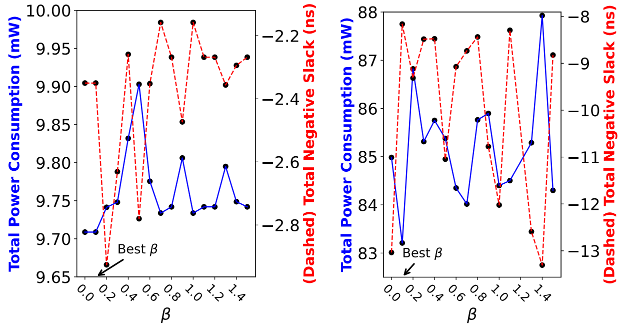
Figure 4: Impact of U on QoR and runtime. Designs: (a) JPEG, (b) VGA.



(a)

(b)

Figure 5: Our heuristic: MBFFs instantiated (as a fraction of the original number of FFs, i.e., L) and average displacement, versus α. Designs: (a) AES, (b) JPEG.



(a)

(b)

Figure 6: Our heuristic: TNS and Total Power Consumption, versus β. Designs: (a) AES, (b) JPEG.

power and TNS as seen for FTray. Thus, we also use $\beta = 0.1$ in the implementation of our method.

Experimental Range Used for the α Parameter. We comment briefly on the large range of α values used in our study. Small values of α generally result in the use of small MBFF sizes (e.g., 2-, 4-bit). Small α values also tend to result in smaller total displacements, since use of a small MBFF will result in lower displacement than use of a large MBFF. In order to activate the usage of larger 8- and 16-bit MBFFs, we must increase α. At the same time, usage of 8-bit MBFFs will normally dominate that of 16-bit MBFFs, since in our ASAP7 enablement 8-bit MBFFs result in very similar power savings compared to 16-bit MBFFs (0.854 vs. 0.850). To avoid having an "inevitable" dominance of 8-bit MBFFs, we include large α values in our study, such that the optimization can potentially leverage the available 0.004 power-per-bit saving.

4.3 Complexity Analysis

We conclude this section with a brief analysis of runtime complexity. (1) **Pointset decomposition.** Assume that each iteration of k-means++ takes T iterations to terminate (convergence). Each call to k-means++ has runtime bounded by $O(T{\cdot}L{\cdot}k)$, where $k \in [2, 8]$. The runtime of our algorithm is bounded by $O(L^2{\cdot}T_{max}{\cdot}k_{max})$, as the size of each sub-problem is variable, and in a pathological scenario L' FFs could be split into two pointsets of $L' - 1$ FFs and 1 FF. Such extreme runtimes are unlikely in real designs, as FFs are usually placed in close proximity, and it is very unlikely to have pathological behavior at all levels of the hierarchical decomposition.[8] (2) **Size of LP and ILP instances.** Our LP and ILP instances have $O(U)$ and $O(U{\cdot}S)$ variables and constraints respectively, where S denotes the number of distinct MBFF sizes. (3) **Size of min-cost flow instances.** Our flow achieves substantial runtime reductions compared to FTray because min-cost flow instances contain only $O(U)$ vertices and $O(U^2)$ edges (i.e., in the complete bipartite graph formed over $O(U)$ FFs and $O(U)$ MBFFs). By contrast, FTray's min-cost flow instances contain $O(L)$ vertices and $O(L^2)$ edges.

5 EXPERIMENTAL SETUP

We now describe the main elements that underlie our experimental validation: (i) codes and tool scripts; (ii) testcases; (iii) modifications to FTray code; (iv) modifications to MShift code; (v) MBFF cell generation; and (vi) evaluation and reporting flows.

Codes and Tool Scripts. Our MBFF clustering approach, including k-means++ based clustering, has been implemented using approximately 3K lines of C++ code. The code invokes Lemon [12] and CPLEX [11] solvers as linked libraries. Lemon [12] is used to solve min-cost flow instances. All LP and ILP instances are solved using CPLEX *v12.10* [11].

Extraction of timing-critical paths after global placement is performed using *Cadence Innovus v21.1*. We extract $20 \cdot L$ FF-to-FF timing-critical paths, where the maximum number of paths per endpoint is 20. We then filter these paths to only consider the top at most 100,000 unique pairs of path start/endpoints, based on slack value (a path that is less critical than another path having the same start/endpoints is dropped from consideration). Values for M in Table 2 give the number of critical paths that are considered in our ILP formulation. Separately, our experiments use single servers with 20 threads; we use a multi-start value of 20 to reflect the number of threads made available in experiments. To ensure reproducibility of our results, all source codes and scripts used in this paper are available in our GitHub repository [13].

Testcases. We test our proposed flow on five designs: AES, MPEG, JPEG and VGA from OpenCores [15], and MemPool Group (MemPool) from ETH Zurich [14]. Table 2 shows for each design the total number of flops (L), total number of timing-critical paths containing clusterable FF endpoints (M), total number of FFs compatible with MBFF library cells, and target clock period (TCP). We use the ASAP7 7nm node technology for our FFs and MBFFs. Specifications per instance are shown in Table 3. As of this writing, the ASAP7 enablement [16] only has MBFFs that are inverting, synchronous,

[8] Recall, perhaps, the worst-case $O(n^2)$ vs. average-case $O(n \log n)$ analysis of Quicksort.

and non-scan. As a result, MemPool contains many FFs that are ineligible for clustering; there are only 64K clusterable FFs out of 360K total FFs in this testcase.

Table 2: Design information. L = number of clusterable FFs. M = number of critical launch-capture paths.

Design	AES	MPEG	JPEG	VGA	MemPool
L	530	3513	4420	17050	64640
M	2905	22824	21863	57257	100000
TCP (*ns*)	0.28	0.20	0.27	0.20	2.30

Modifications of FTray. We rewrite the original FTray implementation, which has been shared by its authors [10] [5], to avoid unnecessary $O(L^2)$ computation and to maintain linear memory usage during the ILP stage.[9] FTray uses capacitated k-means at two stages: once during multi-start and once after multi-start. We run capacitated k-means until convergence or until a maximum of 3 iterations at the multi-start stage. We run capacitated k-means until convergence or until a maximum of 7 iterations after the multi-start stage. This 3-7 split mimics the 30%-70% split mentioned in [4]. We set these maximum numbers of iterations based on a parameter study: exceeding 10 iterations yields little or no change in clustering solution quality, while incurring costly runtime overhead.[10] As noted earlier (Subsection 4.2), we run FTray with our ASAP7-calibrated set of α parameters ($\{4, 67, 130, 193, 264\}$). We report the run yielding the lowest observed total power consumption.

Modifications of MShift. We use the open-source MShift implementation for our experiments [18]. We set the following (MShift-specific) parameters for our runs: $\delta = 0.001$ μm, $\epsilon = 2.5$ μm, $M = \{1, 5, 10, 15, 20\}$, capacity (stable matching) = 1000. Here, we increase the default capacity value in stable matching because some of our test runs failed to return a solution for the default value of 100.[11] Our scripts for netlist modification include a modified TCL script from the IEEE RDF-2020 CEDA repository [10], and a post-processing Python script to ensure that cluster labels are distinct. Because MShift provides no information regarding the assignment of individual FFs to MBFF slots, we assign each FF to the first slot available in its corresponding MBFF.

MBFF Cell Generation. We find that the available ASAP7 MBFF cells in [16] lack power and area scaling compared to their corresponding single-bit FFs. Therefore, we provide a Python script to generate a power-scaled MBFF cell library based on a reference single-bit FF. Additionally, we provide a Python script to generate LEF files for respective MBFF cells. We do not scale the area of an MBFF; we only stack single-bit FFs horizontally and vertically to generate the target MBFFs. Table 3 details the scaling information, including power-per-bit and aspect ratios for the generated MBFFs.

Table 3: MBFF attributes.

MBFF Size	1	2	4	8	16
Normalized Power-Per-Bit	1.00	0.900	0.875	0.854	0.850
Normalized Area-Per-Bit	1	1	1	1	1
Height × Width	1 × 1	2 × 1	4 × 1	4 × 2	8 × 2

Evaluation Flow. We first synthesize each design with the given target clock period (TCP) in Table 2, using *Cadence Genus v21.1*. For our place and route runs, we use *Cadence Innovus v21.1*. We first run placement and use the placed design as input to our MBFF clustering flows to generate an MBFF-clustered netlist and DEF file. Using the updated netlist and DEF file, we run incremental placement, clock tree synthesis, routing, and post-routing optimization (PRO). At the beginning of the evaluation flow, we set the *set_dont_touch* attribute as *size_ok* for all FFs. This ensures that gate sizing and VT swapping can be performed for all FFs, while preventing MBFFs from converting to single-bit FFs (and vice versa), during optimization steps. We report the power, standard cell area, routed wirelength, and the worst and total negative slack (WNS, TNS) after Innovus finishes its PRO step. To generate the baseline results, we skip the MBFF clustering step and capture the PRO metrics. We run all the MBFF clustering flows on a CentOS 8 machine with 2.25 GHz AMD EPYC 7742 64-Core processor and 512GB of memory. Each flow is allowed a maximum of 20 threads.

6 EXPERIMENTAL RESULTS

We now present and discuss our experimental results. We first show comparisons with a baseline (no MBFF clustering) flow and with current SOTA MBFF clustering flows (FTray and MShift). We then examine scalability of our method in terms of multithreading.

6.1 Comparisons with Baseline and Other MBFF Clustering Flows

Table 4 presents the PRO results (from *Cadence Innovus v21.1*) for each method used in this study: *Our* (our proposed method), *MShift*, and *FTray*, across all designs with the exception of MemPool.[12] Each flow occupies 2 rows per design in the table, with subscripts indicating the maximum MBFF size made available to the flow. Specifically, the subscript 16 means that all MBFFs are available to use (i.e., 2-, 4-, 8-, and 16-bit MBFFs). The subscript 4 means that only 2- and 4-bit MBFFs are available to use, reflecting recent methodology trends in industry.

For the experiments that are restricted to 2- and 4-bit MBFFs, we use $\alpha = \{4, 18, 32, 56, 70\}$ for both Alg1 and FTray, as neither flow requires large α values to place 2- and 4-bit MBFFs.[13] For each design, we highlight in bold colored font the best metric achieved across all flows. Red highlights indicate best 4-bit results, and blue highlights indicate best 16-bit results. Table 5 summarizes the results of Table 4. The table displays the number of designs in which a given flow generates the best value for each specific metric.

[9] In the implementation of [4], $O(L^2)$ binary variables are created during the ILP stage (that is, all possible $B_{l,i,j}$ are created). This is unnecessary as $B_{l,i,j}$ can only be turned on if previously turned on (i.e., assigned flow = 1) in the minimum-cost flow solution, which ensures that there are at most L $B_{l,i,j}$ values.

[10] We retain the FTray authors' speedup by restricting the maximum edge length considered in the min-cost flow. However, we drop this restriction in the implementation of our new heuristic: the restriction can cause some MBFFs to never be placed since they may have unused slots, and this creates additional power overhead.

[11] Consider a run of stable matching on L FFs. If there exists a subset F' such that each $f_l \in F'$ has the same potential MBFF assignments, MShift will not return a solution if $|F'| > capacity \cdot max(S_i)$, where $max(S_i)$ is the maximum number of slots in an MBFF.

[12] Due to high runtime and memory usage of Innovus detailed routing for MemPool, we report PPA metrics after global routing for this testcase. Also, due to excessive runtime and memory usage, we could not generate FTray clustering solutions for MemPool.

[13] Recall from Section 4.2 that we use large values of α to enable instantiation of 16-bit MBFFs in the ASAP7 enablement. When 8- and 16-bit MBFFs are made unavailable for use, we can relax our range of α values.

MShift Comparisons. We compare the results of our flow to the results of MShift.

- **Total Power.** When using up to 4-bit MBFFs and when using up to 16-bit MBFFs, our flow produces lower total power consumption on 4 out of 5 designs. When using up to 4-bit MBFFs, our flow achieves a maximum (resp. average) improvement in total power consumption of 5.24% (1.69%). When using up to 16-bit MBFFs, our flow achieves a maximum (resp. average) improvement in total power consumption of 6.18% (1.90%).
- **Timing.** When using up to 16-bit MBFFs, our flow produces improved TNS on 3 out of 5 designs. When using up to 4-bit MBFF usage, our flow improves TNS for only one design.
- **Area and Wirelength.** When using up to 16-bit MBFFs, our flow produces lower area for 3 out of 5 designs and lower wirelength for all 5 designs. On the other hand, when using up to 4-bit MBFFs, our flow produces lower wirelength for only one design and higher area on all designs with an average area increase of 0.45%.
- Overall, we see that our flow is generally superior to MShift in terms of total power consumption both when using up to 4-bit MBFFs and when using up to 16-bit MBFFs. Our flow also shows superiority with regard to area, wirelength and timing but only when using up to 16-bit MBFFs.

FTray Comparisons. We also compare the results of our flow to the results of FTray.

- Our flow results higher total power consumption compared to FTray. This is not surprising since our solution space for MBFF clustering is limited compared to the search space utilized by FTray. Overall, we observe an average degradation of 0.66% (resp. 1.12%) in total power consumption when using up to 4-bit (resp. up to 16-bit) MBFFs.
- Our flow is notably runtime-and memory-friendly when compared to FTray. Specifically, we observe approximately 400× speedup in our flow as compared to FTray. Additionally, we note that FTray uses approximately 435GB of memory to generate a clustering solution for VGA; our flow uses only 2GB of memory to generate a clustering solution for the same VGA testcase.
- When using up to 4-bit MBFFs, our flow achieves lower area and lower wirelength for 2 out of 4 designs. When using up to 16-bit MBFFs, our flow achieves lower area for 2 out of 4 designs, and lower wirelength for 3 out of 4 designs.

Additional Observations. We make a few additional observations regarding the results in Table 4.

- Clock power reduces significantly for all the testcases except for MemPool. This is because in MemPool, the eligible number of FFs for clustering represent only 18% of the total number of FFs, as shown in Table 2. As a result, the reduction in the number of clock pins is low relative to other testcases. Additionally, since approximately 50% of the total power in MemPool is macro power, the opportunity for total power reduction is low compared to other designs.
- Combinational power increases in most cases as a result of the MBFF clustering, since FF movement (i.e., D) increases wirelength and can cause congestion and detouring. As a result, the P&R tool adds more buffers in datapaths, leading to increased combinational power. This can be design-specific, however: MBFF clustering in AES for all flows, and in VGA for MShift, leads to fewer buffers in datapaths and reduced combinational power.
- The total power improvement due to MBFF clustering for AES and JPEG is limited compared to other designs because the sum of sequential and clock power is much smaller than combinational power. Even though the relative reductions of clock and sequential power for these designs is similar to what is seen for other designs, the overall power improvement is small.

6.2 Scalability in Terms of Multithreading

Last, we show the scaling behavior of our method in terms of multithreading. For this, we run our flow on VGA and MemPool using $\{8, 16, 20\ (default), 24, 32, 40\}$ threads with a fixed $\alpha = 770$. We note that α does not impact our flow's runtimes. Figure 7 shows the reduction of runtime as thread count increases. This supports the scalability of our recursive pointset decomposition-based approach for efficient yet high-quality MBFF clustering.

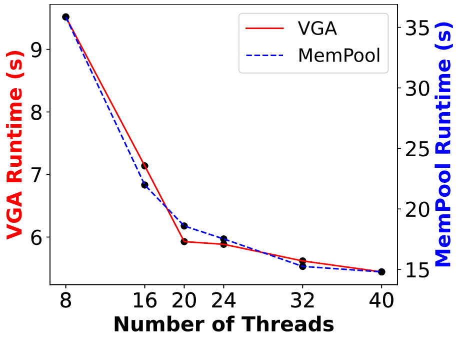


Figure 7: Our Flow: Runtime versus Threads. Designs: VGA, MemPool.

7 CONCLUSION

We have proposed a simple pointset decomposition heuristic to reduce the size of min-cost flow instances arising in the FTray [4] approach to MBFF clustering. We test our proposed method for up to 4-bit and up to 16-bit MBFF clustering on five designs and evaluate the resulting solutions using a commercial P&R tool (*Cadence Innovus v21.1*) flow. Our method achieves up to 400× speedup compared to FTray, while producing slightly worse solution quality. Compared to MShift, our method achieves up to 6.18% reduction in total power, while also achieving improved total negative slack and wirelength for our testcases. Our method produces high-quality results in reasonable runtime for large testcases, and we also demonstrate scalability in terms of multi-threading. Our codebase and scripts are permissively open-sourced in GitHub [https://github.com/ABKGroup/MBFFClustering]. Additionally, our flow has been integrated into OpenROAD [17].

ACKNOWLEDGMENTS

We thank the authors of FTray [4] and of MShift [2] for sharing their implementations [10] [5]. This work is supported in part by DARPA HR0011-18-2-0032 (OpenROAD). We thank Mr. Matt Liberty for guidance on the implementation of our framework and Mr. Dooseok Yoon for providing the initial ASAP7 MBFFs.

Table 4: P&R Results using up to 4-Bit MBFFs ($*_4$) and up to 16-Bit ($*_{16}$) MBFFs. *FT* = FlopTray; *MS* = MeanShift.

Design	Flow	Area (Norm.) (μm^2)	WL (Norm.) (μm)	Timing (ps)		Power (mW)				Runtime (s)
				WNS	TNS	Total (Norm.)	Seq.	Comb.	Clk.	
AES	NC	1580.18 (1.000)	54330 (1.00)	-33	-3560	10.55 (1.00)	3.453	6.348	0.7092	NA
	FT_{16}	1581.73 (1.001)	**52701 (0.970)**	-24	-2582	**9.71 (0.920)**	3.4	6.309	**0**	2.207
	FT_4	1577.51 (0.998)	**52467 (0.966)**	**-30**	**-2691**	9.75 (0.924)	3.253	6.281	**0.2183**	1.517
	MS_{16}	**1578.82 (0.999)**	53264 (0.980)	-27	-2601	9.74 (0.923)	**3.229**	**6.303**	0.2071	0.050
	MS_4	**1575.33 (0.997)**	52949 (0.975)	-31	-3068	9.92 (0.940)	3.276	6.291	0.3519	0.050
	Our_{16}	1589.42 (1.006)	53020 (0.978)	**-22**	**-2219**	9.77 (0.926)	3.296	6.366	0.1088	4.536
	Our_4	1576.45 (0.998)	53579 (0.986)	-31	-2778	**9.73 (0.922)**	**3.23**	**6.279**	0.2196	1.789
MPEG	NC	2045.11 (1.000)	51415 (1.00)	-49	-6088	43.24 (1.00)	28.05	7.688	7.501	NA
	FT_{16}	**2051.45 (1.003)**	54461 (1.059)	-47	**-6459**	**35.48 (0.821)**	**26.26**	**7.951**	**1.275**	101.268
	FT_4	**2041.49 (0.998)**	52524 (1.021)	-49	**-6243**	**35.56 (0.822)**	**25.42**	**7.717**	2.43	63.671
	MS_{16}	2069.86 (1.012)	55750 (1.084)	-49	-7248	37.64 (0.870)	26.94	8.6	2.102	0.230
	MS_4	2045.12 (1.000)	52729 (1.026)	**-47**	-6462	38.16 (0.883)	26.27	7.788	4.1	0.160
	$Ours_{16}$	2052.38 (1.003)	**52594 (1.023)**	**-46**	-6696	37.61 (0.870)	26.42	7.97	3.219	4.652
	$Ours_4$	2051.11 (1.003)	**52299 (1.017)**	-48	-6877	36.16 (0.836)	25.86	8.031	**2.276**	1.929
JPEG	NC	5867.09 (1.00)	108797 (1.00)	-13	-1045	85.04 (1.00)	28.47	48.56	8.011	NA
	FT_{16}	6315.31 (1.076)	121397 (1.116)	-40	-10732	83.64 (0.984)	27.3	54.71	**1.635**	153.948
	FT_4	6261.67 (1.067)	121727 (1.119)	-45	-7351	82.66 (0.972)	**26.45**	53.6	**2.61**	72.266
	MS_{16}	6237.35 (1.063)	121528 (1.117)	-36	-8423	83.89 (0.986)	27.23	53.47	3.199	0.250
	MS_4	**5953.83 (1.015)**	**113732 (1.045)**	**-15**	**-995**	**81.44 (0.958)**	26.75	**50.06**	4.631	0.190
	Our_{16}	**5909.00 (1.007)**	**112089 (1.030)**	**-12**	**-1061**	**80.93 (0.952)**	**27.11**	**49.19**	4.628	4.563
	Our_4	6048.02 (1.031)	116735 (1.073)	-21	-3436	82.89 (0.975)	26.57	52.53	3.782	2.206
VGA	NC	9683.99 (1.00)	273443 (1.00)	-18	-29182	223.70 (1.00)	163.8	23.08	36.79	NA
	FT_{16}	9667.84 (0.998)	301184 (1.101)	-48	**-77661**	**172.84 (0.773)**	146.4	23.51	**4.617**	1763.872
	FT_4	**9603.25 (0.992)**	287572 (1.052)	-50	-119382	**182.78 (0.817)**	**148.7**	22.09	**12.02**	1309.888
	MS_{16}	9653.59 (0.997)	297154 (1.087)	**-47**	-119145	186.22 (0.832)	151.8	23.38	11.03	1.110
	MS_4	9614.18 (0.993)	**285999 (1.046)**	**-37**	**-57320**	193.71 (0.866)	151.7	**21.63**	20.37	0.850
	Our_{16}	**9646.27 (0.996)**	**293481 (1.073)**	-68	-180833	174.71 (0.781)	**144.7**	**22.77**	7.268	6.044
	Our_4	9641.62 (0.996)	288782 (1.056)	-80	-119808	187.84 (0.837)	152.2	22.9	12.74	3.215
MemPool[12]	NC	320348.09 (1.000)	28320704 (1.000)	-2	-55	963.99 (1.000)	197.1	246.4	54.76	NA
	MS_{16}	**319865.94 (0.998)**	28415252 (1.003)	**-2**	**-44**	956.15 (0.992)	**195.7**	**246.9**	47.85	4.080
	MS_4	**319711.13 (0.998)**	**28320005 (1.000)**	**-2**	**-28**	958.00 (0.994)	195.6	**246.0**	50.61	3.550
	Our_{16}	320176.86 (0.999)	**28388492 (1.002)**	**-2**	-52	**955.76 (0.991)**	196.2	247.0	**46.84**	18.597
	Our_4	320384.52 (1.000)	28369933 (1.002)	**-2**	-124	**956.09 (0.992)**	**195.1**	247.4	**47.92**	13.645

Table 5: Number of designs in which a given flow yields the best observed result for each specific metric.

Metric	Max. Size	Flow		
		FTray	Ours	MShift
Power	16-bit	3	2	0
	4-bit	2	2	1
Area	16-bit	1	2	2
	4-bit	2	0	3
Wirelength	16-bit	1	4	0
	4-bit	1	1	3
TNS	16-bit	2	2	1
	4-bit	2	0	3

REFERENCES

[1] D. Arthur and S. Vassilvitskii, “k-means++: The Advantages of Careful Seeding”, *Proc. SODA*, 2007, pp. 1027-1035.

[2] Y.-C. Chang, T.-W. Lin, I. H.-R. Jiang and G.-J. Nam, “Graceful Register Clustering by Effective Mean Shift Algorithm for Power and Timing Balance”, *Proc. ISPD*, 2019, pp. 11-18.

[3] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy and G. Yeric, “ASAP7: A 7-nm FinFET Predictive Process Design Kit”, *Microelectronics J.* 53 (2016), pp. 105-115.

[4] A. B. Kahng, J. Li and L. Wang, “Improved Flop Tray-Based Design Implementation for Power Reducation”, *Proc. ICCAD*, 2016, pp. 20:1-20:8.

[5] J. Li, *personal communication*, 2023.

[6] P. J. Rousseeuw, “Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis”, *J. of Computational and Applied Mathematics* 20 (1987), pp. 53-65.

[7] K. R. Shahapure and C. Nicholas, “Cluster Quality Analysis Using Silhouette Score”, *Proc. DSAA*, 2020, pp. 747-748.

[8] G. Wu, Y. Xu, D. Wu, M. Ragupathy, Y.-Y. Mo and C. Chu, “Flip-flop Clustering by Weighted K-Means Algorithm”, *Proc. DAC*, 2016, pp. 1-6.

[9] ASAP7 PDK and Cell Libraries. https://github.com/The-OpenROAD-Project/asap7

[10] DATC Robust Design Flow. https://github.com/ieee-ceda-datc/RDF-2020/tree/master

[11] IBM ILOG CPLEX. https://www.ibm.com/products/ilog-cplex-optimization-studio

[12] LEMON. https://lemon.cs.elte.hu/trac/lemon

[13] MBFF Clustering. https://github.com/ABKGroup/MBFFClustering

[14] MemPool Repo. https://github.com/pulp-platform/mempool

[15] OpenCores. https://opencores.org/

[16] OpenROAD Flow Scripts. https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts, commit hash: 731787b.

[17] OpenROAD. https://github.com/The-OpenROAD-Project/OpenROAD, commit hash: db8cd96.

[18] Register Clustering. https://github.com/waynelin567/Register_Clustering, commit hash: bad8f27.