An Open-Source Constraints-Driven General Partitioning Multi-Tool for VLSI Physical Design

Ismail Bustany†, Grigor Gaspanyan‡, Andrew B. Kahng, Ioannis Koutis†, Bodhisatta Pramanik, Zhiang Wang
University of California San Diego, La Jolla, CA, USA
†Advanced Micro Devices, San Jose, CA, USA
‡New Jersey Institute of Technology, Newark, NJ, USA
Email: †{ismail.bustany, grigor.gaspanyan}@amd.com, {abk, bopramanik, zhw033}@ucsd.edu, ‡ikoutis@njit.edu

Abstract—With the increasing complexity of IC products, large-scale designs must be efficiently partitioned into multiple blocks, tiles, or devices for concurrent backend place-and-route (P&R) implementation. State-of-the-art partitioners focus on balanced min-cut without considering constraints such as timing or heterogeneity of resource types. They are thus increasingly unsuitable for current physical design requirements. We introduce TritonPart, the first open-source, constraints-driven partitioning tool for VLSI physical design. TritonPart employs efficient algorithms to handle constraints, including multi-dimensional balance, embedding, and timing constraints. Our experimental work affirms its benefits. For standard min-cut partitioning, TritonPart outperforms hMETIS [17], with improvements of up to ∼20% on some benchmarks. For embedding-aware partitioning, TritonPart effectively leverages the embeddings generated by SpecPart [4] and improves upon it by ∼2%. For timing-aware partitioning, TritonPart significantly reduces the number of cuts on timing-critical paths and prevents timing-noncritical paths from becoming critical (−21X, ∼119X reduction relative to hMETIS and KaHyPar [31], respectively).

Index Terms—hypergraph partitioning, multi-dimensional weights, timing, embedding, VLSI constraints

I. INTRODUCTION

Hypergraph partitioning is a fundamental optimization problem in VLSI CAD. Partitioning is arguably becoming even more crucial due to system performance requirements, the more challenging hierarchy of system interconnects, and the scale of modern systems. For backend place-and-route (P&R), large designs must be partitioned into multiple blocks/tiers/devices that achieve timing closure when implemented in parallel or concurrently.

In the standard “one-dimensional” formulation of balanced hypergraph partitioning, each vertex is associated with a single scalar weight. However, in ASIC/FPGA flows, a netlist with different types of elements (e.g. flip-flops, LUTs, DSPs, etc.) can be modeled as a hypergraph where vertices are associated with vectors of weights that give rise to multi-dimensional balance constraints. Moreover, the standard formulation does not take into account timing constraints. Publicly available partitioners such as SpecPart [4], hMETIS [17], and KaHyPar [31] solve the standard formulation, but are not suitable for modern applications such as multi-FPGA partitioning [13], [33], [34] and timing-driven partitioning [11]. In light of the above, there is a need for a 21st-century partitioning multi-tool that should be:

a. Timing-aware and able to handle multi-dimensional weights and balance constraints, and other useful types of constraints.
b. Permissively open-source, easy-to-use, and scalable in order to accommodate future – not just today’s – problem instances.

This paper describes TritonPart, an open-source, constraints-driven, general partitioning framework. TritonPart is designed to address hypergraph partitioning problems under user-specified constraints of multiple types. It is applicable to both classical hypergraph partitioning and timing-aware netlist partitioning. The main contributions of this paper are as follows:

• Hypergraph Partitioning with Pragmatic Constraints:
  We present a generalized hypergraph partitioning formulation. We extend the standard scalar vertex weights to multi-dimensional vectors, with corresponding multi-dimensional balance constraints [4]. We also include in our formulation constraints that are informed by real applications across various domains. These include timing path cut, fixed vertex, grouping [37], and “soft” embedding constraints.

• General Partitioning Multi-Tool:
  We present TritonPart, the first open-source framework that is able to simultaneously honor all the aforementioned constraints. TritonPart is released under a permissively open-source license enabling other researchers to readily adapt it to accommodate different constraints [4].

• Novel Algorithms for Multilevel Partitioning:
  TritonPart follows the established multilevel partitioning paradigm, but also incorporates elements from [4]. In addition, TritonPart consists of multiple algorithms that enable it to handle constraints, including a novel slack propagation algorithm that enables it to reduce the number of cuts on timing-critical paths and prevent timing-noncritical paths from becoming critical.

• An Extensive Experimental Study:
  We evaluate TritonPart against state-of-the-art partitioners (SpecPart [4], hMETIS [17], and KaHyPar [31]), using the Titan23 Suite [25] benchmarks. For the classical hypergraph partitioning problem, on some benchmarks, TritonPart can substantially improve the cutsize by more than 20% compared to state-of-the-art partitioners. We also validate TritonPart’s timing-aware partitioning capabilities on modern VLSI benchmarks with up to 10M instances from the MacroPlacement repository [57]. Our experimental results demonstrate that TritonPart can substantially reduce the number of cuts on timing-critical paths and prevent timing-noncritical paths from becoming critical (∼21X, ∼119X reduction compared to hMETIS and KaHyPar, respectively on some benchmarks).

A. Related Work

Hypergraph partitioning has been extensively studied in past decades, with numerous high-quality partitioners proposed throughout the literature. The majority of these partitioners address the balanced min-cut hypergraph partitioning objective, while some timing-driven partitioners have also appeared.

Min-cut partitioners follow the multilevel paradigm that entails (i) multilevel coarsening that iteratively clusters the input hypergraph to generate a multi-level hierarchy of progressively coarser hypergraphs; (ii) initial partitioning that partitions the coarsest hypergraph to generate a solution; (iii) multilevel refinement that refines the partitioning
solution at each level of the hierarchy; and (iv) V-Cycle refinement where the partitioning solution is used to drive further iterations of restricted coarsening and multilevel refinement [12]. The partitioning solution is preserved during restricted multilevel coarsening by only clustering vertices that belong to the same block of the partitioning solution. Leading partitioners MLPart [6], PtoH [7], BiPart [24], KalyPar [31], and hMETIS [17] follow the multilevel paradigm and are widely used in standard industrial pipelines. Recently, a new supervised spectral-based hypergraph partitioner, SpecPart [4] has been proposed that (i) leverages a partitioning solution as a hint, i.e., supervision; (ii) produces embeddings by incorporating supervision; and (iii) generates high-quality partitioning solutions from the embeddings.

Timing-driven partitioners fall into two categories. (i) Path-based methods attempt to prevent cutting of timing-critical paths. Since there are an exponential number of timing paths in a netlist, path-based approaches focus on a given set P of most critical paths [1, 10, 3, 28, 19]. (ii) Net-based methods define a criticality value (e.g., slack) for each net (hyperedge) which indicates the potential negative impact on timing if the net is cut [22, 8, 10]. However, the net-based approach can result in multiple cuts on some critical paths [1]. Both path-based and net-based approaches focus on minimizing path delays for the top P critical paths, and do not explicitly account for the potential of non-critical paths to become critical due to partitioning. Additionally, to the best of our knowledge, there is no open-source software or executable for timing-driven partitioning available to researchers.

II. PRELIMINARIES

A. Problem Formulation

The input is a hypergraph \( H(V, E) \) where each vertex \( v \in V \) is associated with a non-negative \( m \)-dimensional weight vector \( w_v \), and each edge \( e \in E \) is associated with an \( n \)-dimensional weight vector \( w_e \). We are also given a positive integer \( K \), and want to partition \( H \) into \( K \) blocks. At a high level, given all inputs \( \mathbf{I} \), we want to compute a partition of \( V \) into \( K \) disjoint blocks \( S = \{V_1, \ldots, V_K\} \) that minimizes a cost function:

\[
\Phi(I, S) = \Phi_{\text{cut}}(I, S) + \Phi_{\text{time}}(I, S)
\]

where \( \Phi_{\text{cut}}(I, S) \) measures the cutsizes as in the standard formulation of hypergraph partitioning and \( \Phi_{\text{time}}(I, S) \) captures the timing cost.

Our generalized formulation of balanced hypergraph partitioning is able to accommodate multiple constraints, each associated with a set of optional inputs (Table I). The constraints we consider are motivated by multiple applications.

- **Fixed vertex constraint:** If \( v \in V_i \), then vertex \( v \) must be in block \( V_i \) in the output partition \( S \). One application of this constraint is the reassignment of certain IP modules to specific blocks prior to partitioning. This is a hard constraint.

- **Grouping constraint:** Vertices that belong to the same group should be assigned to the same block in \( S \):

\[
\text{block}(v) = \text{block}(u) \text{ if } \text{Cmty}(v) = \text{Cmty}(u) \text{ and } \text{Cmty}(v) > 1
\]

Here each community is assigned a unique index greater than 1 and \( \text{Cmty}(v) = 1 \) signifies the special case when \( v \) is not part of a group. A practical example of this constraint is ensuring that closely-related standard-cell logic connected to the same macro remains together during the partitioning process [23]. Such closely-related standard-cell logic would be assigned to its own group with a unique index. This is a hard constraint.

- **Multi-dimensional balance constraint:** Let \( w_v(j) \) denote the \( j^{th} \) coordinate of \( w_v \). We define \( W_j = \sum_{v \in V} w_v(j) \). Then we require that the solution \( S \) satisfies the following for all \( 1 \leq i \leq K, 1 \leq j \leq m \):

\[
\left( \frac{1}{K} - \epsilon \right) W_j \leq \sum_{v \in V_i} w_v(j) \leq \left( \frac{1}{K} + \epsilon \right) W_j
\]

In other words, the standard balance constraint must be satisfied along each dimension of weights. A practical example of this constraint is partitioning a netlist across multiple FPGAs, where resources such as flip-flops (FFs), digital signal processing blocks (DSPs), and look-up tables (LUTs) are all limited. This is a hard constraint.

- **Embedding constraint:** Vertices that are closer in the embedding \( X \) (see Table I) should preferably be assigned to the same block. This is a soft constraint that is used to inform algorithmic decisions of the TritonPart flow, similar to the idea in [22]. The embedding constraint can be based on real placement locations generated by a placer in the physical design flow or be generated by any embedding algorithm, including spectral methods such as SpecPart [4].

- **Timing constraint:** The sets of paths \( P, P_{\text{non}} \) and their associated slacks (see Table I), can come as input from a static timing analyzer (e.g., OpenSTA [26]). We want the partition to minimize the number paths in \( P \) that are cut. For a \( K \)-way partition, we also want to control the maximum number of cuts on any timing-critical path. Additionally, we want to control the number of timing-noncritical paths \( P_{\text{non}} \) that turn critical after partitioning. Essentially, for each timing-noncritical path \( p_{\text{non}} \in P_{\text{non}} \), we measure timing-criticality after introducing an extra delay \( \Delta \) for each time \( p_{\text{non}} \) is cut. If the total delay increment incurred on \( p_{\text{non}} \) exceeds its slack, \( \text{slack}(p_{\text{non}}) \), then we say that \( p_{\text{non}} \) has become timing-critical. Given the above, and a user-defined parameter \( \alpha \) (Table I), we define the cutsize (cut cost) as:

\[
\Phi_{\text{cut}}(I, S) = \sum_{e \in E} \langle \alpha, w_e \rangle \text{Cut}(e)
\]

where \( \langle \cdot, \cdot \rangle \) denotes the inner-product. The timing cost is discussed in more detail in Section IV where we define quantities \( t_e, t_p, D(p) \), and \( SF(p) \), listed in Table I. At a high level \( \Phi_{\text{time}}(I, S) \) consists of three
components: (i) the timing cost (Cut) associated with hyperedges that are cut; (ii) the timing cost (D) associated with timing-critical paths that are cut; and (iii) the timing cost (SF) associated with the 

\[ \Phi_{\text{time}}(T, S) = \sum_{e} \beta e \cdot \text{Cut}(e) + \sum_{p \in \mathcal{P}} (\gamma D(p) t_p + \tau SF(p)) \]  

where \( \beta, \gamma, \tau \) are user-defined scalar parameters that control the relative importance of the corresponding costs (see Table I).

III. OUR APPROACH

In this section, we discuss the flow of the proposed constraints-driven general hypergraph partitioning framework, called TritonPart. We have released TritonPart with a permissively open-source license at [20]. Similar to hMETIS [17] and KaHyPar [31], TritonPart adopts a multilevel framework for handling large-scale hypergraphs. However, TritonPart distinguishes itself with two major differences. (i) It can concurrently handle multiple constraints such as fixed vertex constraint, multi-dimensional balance constraint, grouping constraint, embedding constraint, and timing constraint. (ii) It integrates the cut-overlay clustering and partitioning techniques from [3] into the multilevel framework. The flow of TritonPart is illustrated in Figure 1 with details given in Algorithm 1.

![Fig. 1: Flow of the TritonPart framework.](image)

Lines 5-6: We generate multiple orderings of the vertices in \( H \). Each ordering induces a unique multilevel hierarchy of coarser hypergraphs using constraints-driven coarsening. [Section III-A]

Lines 7-14: For each hierarchy, we apply an initial partitioning and refinement step. This generates multiple candidate partitioning solutions \( S_1, ..., S_6 \) where \( \theta \) is an input parameter. [Sections III-B and III-C]

Line 18: The candidate solutions are utilized by the cut-overlay clustering and partitioning algorithm to generate a much better partitioning solution \( S' \). [Section III-D]

Lines 19-29: V-Cycle refinement uses \( S' \) to further optimize the cost function in Equation (1) and generate the output partitioning solution \( S \) that satisfies all hard constraints. We emphasize that our V-Cycle refinement is different from that of a standard multilevel partitioner. Notably, we run an additional step of ILP-based partitioning on the coarsest hypergraph \( H_\zeta \) if the number of vertices in \( H_\zeta \) is less than a threshold \( \text{thr}_\text{ilp} \). [Section III-E]

The following sections elaborate on the components of TritonPart.

A. Constraints-Driven Coarsening

In the multilevel partitioning paradigm, the first step involves multilevel coarsening, which constructs a sequence of progressively coarser hypergraphs. More specifically, at each level, clusters of vertices are identified, and then merged and represented as a single vertex in the resulting coarser hypergraph [17]. One of the most effective coarsening schemes is First-Choice (FC) [17], [31], which traverses the vertices in the hypergraph according to a given ordering and merges pairs of vertices with high connectivity. The connectivity between a pair of vertices \((u, v)\) is measured using the heavy-edge rating function [31]:

\[ r(u,v) = \sum_{\epsilon \in \{l(u) \cap l(v)\}} \frac{\langle \alpha, w_\epsilon \rangle}{|\epsilon| - 1}. \]  

However, the FC scheme is not directly applicable when we address the multiple constraints present in the general hypergraph partitioning problem. To efficiently manage these constraints, we propose the following enhancements to the FC scheme.

Algorithm 1: TritonPart framework.

Input: Standard Inputs: \( H, K, \epsilon \)
Constraint Inputs: \( V_C, Cmty, P, P_{\text{non}}, \text{slack}_p, \Delta, X \)
Additional Parameters: \( \theta, \eta, \eta_p, \text{thr}_\text{ilp} \)
Output: Partitioning solution \( S \)

1. merge all the vertices in the same group into one vertex and update the hypergraph \( H \)
2. merge all the vertices that are preassigned into the same block into one vertex and update the hypergraph \( H \)
3. create an empty list \( S_{\text{candidate}} = \{\} \) for candidate solutions
4. for \( i = 1; i \leq \theta; i + + \) do
5. /* Coarsening [Section III-A] */
6. generate random ordering of vertices
7. \( \{H_c\} = \{H_{c1}, H_{c2}, ..., H_{c_\theta}\} \) ← perform constraints-driven First-Choice-based coarsening using the ordering to generate a hierarchy of successively coarser hypergraphs
8. /* Initial Partitioning [Section III-B] */
9. \( S_{\text{init}} \) ← perform initial partitioning heuristics
10. \( S_{\text{init}} \) ← pick the best \( \eta_p \) solutions from \( S_{\text{init}} \)
11. /* Parallel Refinement [Section III-C] */
12. while \( \{H_c\}.\text{empty}() == \text{false} \) do
13. \( H_\epsilon \leftarrow \{H_c\}.\text{back}() \); \( \{H_c\}.\text{pop_back}() \)
14. /* All the solutions in \( S_{\text{init}} \) are refined in parallel */
15. for each solution \( S_j \in S_{\text{init}} \) do
16. \( \tilde{S}_j \) ← perform refinement heuristics on \( H_\epsilon \) and \( S_j \)
17. end
18. end
19. /* Cut-overlay clustering and partitioning [Section III-D] */
20. \( S' \) ← perform cut-overlay clustering and ILP-based partitioning on \( S_{\text{candidate}} \)
21. /* V-Cycle refinement guided by \( S' \) [Section III-E] */
22. \( \{H_c\} = \{H_{c1}, H_{c2}, ..., H_{c_\theta}\} \) ← perform constraints-driven First-Choice-based coarsening guided by \( S' \) to generate a hierarchy of successively coarser hypergraphs
23. if \( H_{c_\theta}.\text{num_vertices} \leq \text{thr}_\text{ilp} \) then
24. \( S \) ← perform ILP-based partitioning on \( H_{c_\theta} \)
25. else
26. while \( \{H_c\}.\text{empty}() == \text{false} \) do
27. \( H_\epsilon \leftarrow \{H_c\}.\text{back}() \); \( \{H_c\}.\text{pop_back}() \)
28. perform refinement heuristics on \( H_\epsilon \) and \( S \)
29. end
30. return \( S \)
• **Fixed vertex constraint:** Fixed vertices that belong to the same partitioning block are merged into a single vertex. This approach respects the immobility of these vertices and prevents the coarsening process from violating these constraints.

• **Grouping constraint:** Vertices that belong to the same group are merged into a single vertex. This enforces the constraint of shared group membership.

• **Embedding constraint:** The embedding information is incorporated into the heavy-edge rating function, which is updated to

\[
\hat{r}(u, v) = r(u, v) + \frac{1}{||X_u - X_v||_2^2},
\]

In Equation (7), \(X_u\) is the embedding of vertex \(u\), \(r(u, v)\) is the score function in Equation (6), and \(\rho\) is a normalization factor. We set \(\rho\) such that the average distance between two vertex embeddings is equal to the average standard rating score \(r(u, v)\) in Equation (6). A pair of vertices \((u, v)\) that are in close proximity within the embedding space are assigned a higher rating score. This consideration mirrors the intuitive concept that closely embedded vertices share a stronger connection. Recall that vertices are merged by our constraints-driven coarsening framework. When vertices \(v_1, \ldots, v_t\) are merged into a single vertex \(v_{\text{coarse}}\), we define the embedding \(X_{v_{\text{coarse}}}\) of \(v_{\text{coarse}}\) to be the center of gravity of the \(t\) vertices, i.e., the following convex combination:

\[
X_{v_{\text{coarse}}} = \sum_{j=1}^{t} \frac{||w_{v_j}||}{M} X_{v_j}, \quad \text{where} \quad M = \sum_{j=1}^{t} ||w_{v_j}||.
\]

• **Community guidance:** Standard multilevel coarsening algorithms have the ability to perform community-guided coarsening where only vertices within the same community are considered for merging [12]. We adopt the same methodology in our constraints-driven coarsening framework. This is used by the cut-overlay clustering and partitioning (Section III-D) and V-Cycle refinement (Section III-F).

• **Tie-breaking mechanism:** If multiple neighbor pairs have the same rating score, we favor combining the lexicographically first unmatched vertex to break ties.

Coarsening in the presence of timing constraints is described in Section [IV-B]. To reduce runtime, we use a “multi-node matching” scheme similar to that of BiPart [24].

**B. Initial Partitioning**

After completing the coarsening process, we find an initial partitioning solution for the coarsest hypergraph \(H_c\). The small size of \(H_c\) enables us to apply various partitioning methods, including computationally demanding ones such as integer linear programming (ILP). We run multiple variants for initial partitioning.

• **Random and VILE partitioning.** As highlighted in [11], the best initial partition of the coarsest hypergraph does not necessarily result in the best partition of the original hypergraph. As a result, we conduct \(\eta\) (50 by default) runs of random initial partitioning using distinct random seeds to ensure a diverse set of initial solutions. In addition to this, we perform a “VILE” partitioning [5] to generate a reasonably good partition of the coarsest hypergraph.

• **ILP-based partitioning.** We also run an ILP to find an initial partitioning of \(H_{c_{\ell}}\). We optimize only the cutsize rather than the cost function \(\Phi\) to reduce the complexity of the ILP formulation. To keep the runtime of TritonPart manageable, we run the ILP-based partitioning only if the number of vertices in the coarsest hypergraph \(H_{c_{\ell}}\) does not exceed a threshold \(thr_{\text{ilp}}\) (default = 50).

To formulate the balanced hypergraph partitioning problem as an ILP, for each block \(V_i\), we introduce integer \(\{0, 1\}\) variables \(x_{v,i}\) for each vertex \(v\) and \(y_{e,i}\) for each hyperedge \(e\). Using the notation from Section [II], we define the following constraints for each \(i \in [1, K]\):

- \(\sum_{v=1}^{K} x_{v,i} = 1\) for each \(v \in V\)
- \(y_{e,i} \leq x_{v,i}\) for each \(e \in E_i\), and each \(v \in e\)
- \(x_{v,i} = 1\) if \(v \in V_i\), i.e., if \(v\) is fixed to block \(V_i\)
- \(W_j = \sum_{e \in V_i} y_{e,i}\) for each \(j \in [1, m]\)

**Multi-dimensional balance constraint:** for \(1 \leq j \leq m\)

\((K^{-1} - \epsilon) W_j \leq \sum_{e \in V_i} y_{e,i} \leq (K^{-1} + \epsilon) W_j\)

Observe that the first two constraints enforce these two requirements:

\[
x_{v,i} = 1 \text{ if } v \in V_i \quad \text{ and } \quad y_{e,i} = 1 \text{ if } e \subseteq V_i
\]

The objective is to maximize the total weight of the hyperedges that are not cut by the partitioning solution, i.e.,

\[
\max \sum_{r \leq e < K} \sum_{e \in E} \langle v, w_e \rangle y_{e,i}
\]

We run our ILP solver to optimality. We speed its runtime through a warm-start scheme. Specifically, we use the best partitioning solution derived from random and VILE partitioning as a starting feasible solution for the ILP solver.

**C. Refinement**

After a feasible solution of \(H_{c_{\ell}}\) is obtained by initial partitioning, we perform uncoarsening and move-based refinement to improve the partitioning solution. These are performed level by level. At each level, three types of refinement heuristics are applied, in sequence.

• **K-way pairwise FM (PM).** K-way pairwise FM addresses multi-way partitioning via concurrent bipartitioning problems in a restricted version of K-way FM [9]. Given \(K\) blocks of a partition, a refinement pass of PM includes the following steps. (i) \([K/2]\) pairs of blocks are obtained, with refinement specific vertex movements restricted to associated paired blocks. In particular, paired blocks are obtained using the gain-based configuration [9]. (ii) Two-way FM [13] is concurrently performed on all the block pairs. (iii) A new configuration of block pairs is computed at the end of the PM pass for subsequent passes. Even though PM often outperforms K-way FM [9], it still explores a subset of the solution space, as it selects \([K/2]\) block pairs out of \(K(K-1)/2\) block pairs in each pass. To mitigate this, we run a direct K-way FM after PM, as described next.

• **Direct K-way FM.** Our implementation uses \(K\) priority queues. Unlike the traditional gain-bucket data structure [17], which only accommodates integer gain values, and Sanchis’s method [30] that relies on \((K-1)\) priority queues, our approach can manage floating-point gain values and significantly larger values for \(K\). Some key implementation details are the following. (i) For each block \(V_i\), we establish a priority queue that stores the vertices that can be potentially moved from their current block to block \(V_i\). This queue is ordered according to the gain of vertices. The gain of a vertex \(v\) is determined by the reduction in cost in Equation (5) when moving \(v\) from its current block to \(V_i\). (ii) After a vertex move, each priority queue is updated independently, thus enabling parallel updates via multi-threading. (iii) To speed up the refinement we employ the early-stop mechanism in [17]. Specifically, we only move a limited number of vertices (100 by default) in each pass. (iv) We mitigate the “corking effect” [6] by traversing the priority
queue belonging to the vertex with the highest gain and identifying a feasible vertex move.

- **Greedy Hyperedge Refinement (HER).** Greedy hyperedge refinement moves groups of vertices belonging to hyperedges that cross the partition boundary, i.e., hyperedges spanning multiple blocks. We apply the HER approach \cite{49} to complement vertex-based refinement approaches (PM and FM) that move a single vertex at a time and can struggle to effectively refine a hypergraph containing multiple hyperedges with a large subset of vertices. HER mitigates this limitation by moving groups of vertices instead of a single vertex. Our HER approach operates as follows. (i) We randomly visit all the hyperedges. (ii) For each hyperedge $e$ that crosses the partition boundary, we determine whether we can move a subset of the vertices in $e$ without violating the multi-dimensional balance constraints. The objective is to make $e$ entirely contained in a block.

We also adapt our multilevel refinement framework\cite{50} to accommodate timing constraints. This is explained in detail in Section IV-C.

**D. Cut-Overlay Clustering and Partitioning (COCP)**

Cut-overlay Clustering and Partitioning (COCP) is a mechanism to combine multiple good-quality partitioning solutions to generate an improved solution \cite{51}. Given $\theta$ candidate solutions ($S_{\theta}$), classical multilevel partitioners like hMETIS pick the best solution and discard the rest. In contrast, we combine all candidate solutions through COCP. We first denote the sets of hyperedges cut in these solutions by $E_1, \ldots, E_{\theta} \subseteq E$. In COCP, we perform the following steps. (i) We remove $\cup_{i=1}^{\theta} E_i$ from the hypergraph $H(V, E)$, resulting in a number of connected components. (ii) We merge all vertices within each connected component to form a coarser hypergraph $H_{\text{overlay}}$. If the number of vertices in $H_{\text{overlay}}$ is less than $\text{thr}_{\text{ilp}}$, we apply ILP-based partitioning [Section III-B]. If not, we conduct a single round of constraints-driven coarsening to further reduce the size of $H_{\text{overlay}}$ and generate a coarser hypergraph $H'_{\text{overlay}}$. In particular, we use the best candidate solution from $S_{\theta}$ to apply community guidance [Section III-A]; this guarantees that the best candidate solution is preserved in $H'_{\text{overlay}}$. (iii) If the number of vertices in $H'_{\text{overlay}}$ is less than $\text{thr}_{\text{ilp}}$, we apply ILP-based partitioning [Section III-B]. If not, we simply retain the best candidate solution. (iv) We perform multilevel refinement to further improve the partitioning solution at each level of the hierarchy [Section III-C] and return the improved solution $S'$.

**E. V-Cycle Refinement**

Cut-overlay clustering and partitioning produces a high-quality partitioning solution $S'$. To further improve $S'$, we adopt V-Cycle refinement and run it for multiple iterations\cite{52}, similar to hMETIS\cite{53}. Our V-Cycle refinement consists of three phases: multilevel coarsening, ILP-based partitioning, and refinement. During the multilevel coarsening phase, we use $S'$ as a community guidance for the constraints-driven coarsening [Section III-A]. In this phase, only vertices within the same block are permitted to be merged. This ensures that the current partitioning solution $S'$ is preserved in the coarsest hypergraph $H_{\text{c}}$. In the ILP-based partitioning phase, if the number of vertices in $H_{\text{c}}$ does not exceed $\text{thr}_{\text{ilp}}$, we run ILP-based partitioning to improve $S'$. If not, we continue with $S'$ in successive iterations of the V-Cycle refinement. The refinement phase is carried out as described in Section III-C. In the presence of timing constraints, we modify the refinement phase as described in Section IV-C.

**IV. TIMING-AWARE NETLIST PARTITIONING**

**TritonPart’s timing-aware partitioning framework combines path-based and net-based methodologies. Traditional path-based and net-based approaches typically focus on optimizing cuts for the top P timing-critical paths, ignoring the potential for noncritical paths to become critical due to partitioning. To address this, we introduce a slack propagation methodology that optimizes cuts for both timing-critical and timing-noncritical paths.**

**A. Extraction of Timing Paths and Slack Information**

**TritonPart** first extracts the top $P$ timing-critical paths and the slack information for each hyperedge, leveraging the wireload model (WLM) from the open-source static timing analyzer OpenSTA \cite{26}. We use the findPathEnds function from SearchLib available at \cite{39}. Here we set group_count ($|P|$), endpoint_count, unique_pins and sort_by_slack to 100000, 1, true and true, respectively. We then calculate the timing cost for cutting a timing path and the timing cost for cutting a hyperedge.

**Timing cost for a path.** The timing cost $t_p$ of a path $p$ is determined by its slack $\text{slack}_{p}$,

$$t_p = (1 - \frac{\text{slack}_{p} - \Delta}{\text{clock_period}})^{\mu} \tag{8}$$

where a fixed extra delay $\Delta$ (whose value is specified in Section V) is introduced for timing guardband, and $\mu$ (default = 2) is the exponent.

**Snaking factor.** The snaking factor \cite{35} of a path $p$, denoted as $SF(p)$, quantifies the extent to which the timing path “snakes” or zigzags its way through various blocks. Specifically, $SF(p)$ is defined as the maximum number of block re-entries along the path $p$. Consider Figure 2 which illustrates two different partitions for the timing path $p$, which consists of FF1 $\rightarrow$ Combinational module A $\rightarrow$ Combinational module B $\rightarrow$ FF2. In Figure 2(a), the blocks $V_1$ and $V_2$ experience re-entries 1, 0 and 0 times respectively, hence $SF(p)$ equals 1. However, in Figure 2(b), each block $V_0$, $V_1$ and $V_2$ is entered only once, thereby resulting in $SF(p)$ being 0. In both cases, the number of cuts on $p$, denoted as $D(p)$, is two. However, the snake factor in Figure 2(b) is lower, which is more desirable from a timing perspective. We consider $SF(p)$ in our cost function defined in Equation 6.

**Timing cost for a hyperedge.** The timing cost $t_e$ of a hyperedge consists of two parts: (i) the timing weight corresponding to the hyperedge’s slack $\text{slack}_{e}$; and (ii) the accumulated timing cost of all paths traversing the hyperedge.

$$t_e = (1 - \frac{\text{slack}_{e} - \Delta}{\text{clock_period}})^{\mu} + \sum_{p \in \epsilon(e)} t_p \tag{9}$$

**B. Timing-aware Coarsening**

**TritonPart**’s timing-aware coarsening builds upon the constraints-driven coarsening framework (Section III-A). We add the timing cost of a hyperedge, $t_e$, to the rating score in Equation 7, so as to merge vertices associated with hyperedges with high timing cost. If vertices $(u, v)$ are associated with multiple critical paths then they are more likely to be merged; this is reflected in our rating function $r_t(u,v)$:

$$r_t(u,v) = r(u,v) + \sum_{e \in \{I(u) \cup O(u)\}} \frac{\beta t_e}{|e| - 1} \tag{10}$$
Our timing-aware refinement is based on the cost function in Equation (5). The cost function in Equation (5) seeks to optimize the timing cost based on the top $P$ paths extracted by OpenSTA. To prevent TritonPart from inadvertently turning noncritical timing paths into critical, we perform an additional slack propagation step at the end of each PM/FM/HER pass. In slack propagation, we continuously manage noncritical timing paths from turning critical by repeatedly updating the slack on all nets (hyperedges) and paths after each refinement pass. Our slack propagation methodology is presented in Algorithm 2 and consists of the following steps.

**Algorithm 2: Slack propagation.**

**Input:** Hypergraph $H(V,E)$, Extra delay $\Delta$, Partitioning solution $S$, Extracted timing paths $P$

1. $E_{cut} \leftarrow$ identify all hyperedges in $H$ cut by $S$
2. for each hyperedge $e \in E_{cut}$
   /* Introduce the extra delay for each cut hyperedge */
   3. $slack_e \leftarrow slack_e - \Delta$
   /* Propagate the introduced extra delay by traversing the timing graph */
   4. $\{p_e\} \leftarrow$ identify all timing paths traversing hyperedge $e$
   5. for each timing path $p \in P$
       6. propagate the delay backward and stop the propagation if the slack of the fanout hyperedge is less than $slack_e$
       7. propagate the delay forward and stop the propagation if the slack of the fanin hyperedge is less than $slack_e$
   8. end
9. end
10. update the slack, $slack_p$ of each timing path $p \in P$
11. update the timing cost, $t_p$ for each hyperedge $e \in E$

**A. Validation of Min-cut Partitioning**

We first assess the min-cut partitioning capability of TritonPart by comparing it to leading min-cut partitioners hMETIS and SpecPart, with their default parameter settings. The default parameters for hMETIS are $Nruns = 10$, $Ctype = 1$, $Rtype = 1$, $Vcycle = 1$, $Reconst = 0$, and $seed = 0$ [14]. For SpecPart, the default parameters are $\delta = 5$, $\beta = 2$, $\gamma = 500$, $\zeta = 2$, $\theta = 40$ and $m = 2$ [4]. We use the Titan23 benchmarks [25] for evaluation, with the benchmark statistics presented in Table II. Given the sensitivity of partitioners such as hMETIS and TritonPart to random seeds, cutsize values reported for hMETIS and TritonPart are the averaged (50 trials) best of 20 runs, i.e., sampling 20 solutions of hMETIS and TritonPart 50 times and reporting averaged best of 20 cutsize (hM20, $TP_{20}$). Table II and Figure 3 present the results for $K = 2, 3, 4$ respectively with $\epsilon = 2\%$, with cutsize values rounded to the nearest integer. In Figure 3 cutsize values are normalized by those obtained from hM20. From Table II and Figure 3 we can draw the following conclusions:

- For $K = 2$, TritonPart generates partitioning solutions that are on average $\sim 4.5\%$ ($\sim 1\%$) better than those of hMETIS (SpecPart).
- For $K = 3$, TritonPart generates partitioning solutions that are on average $\sim 8\%$ better than those of hMETIS, with over $30\%$ improvement for the denoise and gsm \textit{switch} benchmarks.
- For $K = 4$, TritonPart generates partitioning solutions that are on average $\sim 8\%$ better than those of hMETIS, with over $30\%$ improvement for the gsm \textit{switch} and bitcoin \textit{miner} benchmarks.

\[\text{Algorithm 2: Slack propagation.} \]

**Input:** Hypergraph $H(V,E)$, Extra delay $\Delta$, Partitioning solution $S$, Extracted timing paths $P$

1. $E_{cut} \leftarrow$ identify all hyperedges in $H$ cut by $S$
2. for each hyperedge $e \in E_{cut}$
   /* Introduce the extra delay for each cut hyperedge */
   3. $slack_e \leftarrow slack_e - \Delta$
   /* Propagate the introduced extra delay by traversing the timing graph */
   4. $\{p_e\} \leftarrow$ identify all timing paths traversing hyperedge $e$
   5. for each timing path $p \in P$
       6. propagate the delay backward and stop the propagation if the slack of the fanout hyperedge is less than $slack_e$
       7. propagate the delay forward and stop the propagation if the slack of the fanin hyperedge is less than $slack_e$
   8. end
9. end
10. update the slack, $slack_p$ of each timing path $p \in P$
11. update the timing cost, $t_p$ for each hyperedge $e \in E$

**A. Validation of Min-cut Partitioning**

We first assess the min-cut partitioning capability of TritonPart by comparing it to leading min-cut partitioners hMETIS and SpecPart, with their default parameter settings. The default parameters for hMETIS are $Nruns = 10$, $Ctype = 1$, $Rtype = 1$, $Vcycle = 1$, $Reconst = 0$, and $seed = 0$ [14]. For SpecPart, the default parameters are $\delta = 5$, $\beta = 2$, $\gamma = 500$, $\zeta = 2$, $\theta = 40$ and $m = 2$ [4]. We use the Titan23 benchmarks [25] for evaluation, with the benchmark statistics presented in Table II. Given the sensitivity of partitioners such as hMETIS and TritonPart to random seeds, cutsize values reported for hMETIS and TritonPart are the averaged (50 trials) best of 20 runs, i.e., sampling 20 solutions of hMETIS and TritonPart 50 times and reporting averaged best of 20 cutsize (hM20, $TP_{20}$). Table II and Figure 3 present the results for $K = 2, 3, 4$ respectively with $\epsilon = 2\%$, with cutsize values rounded to the nearest integer. In Figure 3 cutsize values are normalized by those obtained from hM20. From Table II and Figure 3 we can draw the following conclusions:

- For $K = 2$, TritonPart generates partitioning solutions that are on average $\sim 4.5\%$ ($\sim 1\%$) better than those of hMETIS (SpecPart).
- For $K = 3$, TritonPart generates partitioning solutions that are on average $\sim 8\%$ better than those of hMETIS, with over $30\%$ improvement for the denoise and gsm \textit{switch} benchmarks.
- For $K = 4$, TritonPart generates partitioning solutions that are on average $\sim 8\%$ better than those of hMETIS, with over $30\%$ improvement for the gsm \textit{switch} and bitcoin \textit{miner} benchmarks.
TABLE II: Cutsize comparisons for hMETIS, SpecPart, and TritonPart on Titan32 benchmarks for $K = 2, 3, 4$ and $\epsilon = 2\%$.

B. Validation of Embedding-aware Partitioning

To evaluate the embedding-aware partitioning capability of TritonPart, we propose a two-step evaluation method. First, we use two-dimensional embeddings generated by SpecPart with its default settings, on the dart, denoise, sparseT1Chip2 and directrf benchmarks. These embeddings consist of the two first nontrivial eigenvectors of a generalized eigenvalue problem [3]. Next, we use these embeddings as input embedding constraints to TritonPart. The results are presented in Table III. Here, $TP_{emb}$ represents the cutsize from a single TritonPart run, while $TP_{emb2}$ represents the best cutsize across twenty runs. $TP_{emb2}$ produces partitioning solutions that outperform those of hMETIS, SpecPart, and TritonPart without embedding constraints by $\sim11\%$, $\sim2\%$, and $\sim8\%$, respectively. These results suggest that with high-quality embeddings as input, TritonPart can effectively utilize the embeddings to generate high-quality partitioning solutions (even better than SpecPart).

TABLE III: Effect of embedding for $K = 2$, $\epsilon = 2\%$.

C. Validation of Timing-driven Partitioning

We now assess TritonPart's timing-driven partitioning capability. We consider the following two categories of timing metrics.

- **Evaluation of cuts on timing-critical paths.** This consists of the average number of cuts on each timing-critical path ($P_{avg\_cut}$) and worst (maximum) number of cuts on any timing-critical path ($P_{wst\_cut}$).

- **Evaluation of cuts on timing-noncritical paths.** This consists of the number of timing-noncritical paths which became critical ($P_{n\_critical}$) due to partitioning, average number of cuts on each timing-noncritical path that became critical ($P_{n\_avg\_cut}$), and worst (maximum) number of cuts on any timing-noncritical path that became critical ($P_{n\_wst\_cut}$).

As mentioned in Section C, there are no open-source timing-driven partitioners available for direct comparison. Toward a fair comparison, we propose reasonable baselines based on hMETIS and

Fig. 4: Timing-driven partitioning evaluation flow.
These results suggest that the timing-driven TritonPart can help avoid timing-noncritical paths becoming critical.

<table>
<thead>
<tr>
<th>Design</th>
<th>$F_{ avg _ cut }$</th>
<th>$F_{ wst _ cut }$</th>
<th>runtime (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aracne (NG)</td>
<td>0.26</td>
<td>0.72</td>
<td>7.5</td>
</tr>
<tr>
<td>BlackParrot (NG)</td>
<td>0.37</td>
<td>0.59</td>
<td>8.8</td>
</tr>
<tr>
<td>MemPostG-(NG)</td>
<td>0.60</td>
<td>1.01</td>
<td>8.9</td>
</tr>
<tr>
<td>MemPostG-(GG)</td>
<td>0.76</td>
<td>1.09</td>
<td>10.5</td>
</tr>
<tr>
<td>Aracne (GG)</td>
<td>0.73</td>
<td>1.04</td>
<td>10.5</td>
</tr>
<tr>
<td>BlackParrot (GG)</td>
<td>0.83</td>
<td>1.08</td>
<td>11.0</td>
</tr>
</tbody>
</table>

TABLE VA: Results on timing-critical paths for $K = 5$ and $\epsilon = 2\%$, $hM$, $KHP$, $TP$, and $T_P$, respectively stand for $hMETIS$, KaHyPar, TritonPart, and timing-driven TritonPart. ['-G': Group, '-C': Cluster, NG: NG45, GF: GF12.]

D. Hyperparameter Selection

TritonPart includes the following four lower-level parameters omitted from Algorithm 1 (i) $\psi$: the coarsen hyperedge size skip: hyperedges of size larger than this threshold are excluded from the constraints-driven coarsening phase. (ii) $\chi$: the coarsening ratio: the maximum ratio between the numbers of vertices of two successive hypergraphs in the multilevel hierarchy. (iii) $\phi$: the maximum number of vertices that can be moved in each pass of the refinement phase. (iv) $\Psi$: the number of candidate solutions $\psi$ generated from a distinct vertex order [Section III].

We have determined default values for these hyperparameters by performing an empirical study with $K = 2$ and $\epsilon = 2\%$, involving 5 benchmarks: LU230, LU100, sparcT1_chip2, directrf, and bitcoin_miner. We define the score value as the average improvement in cutsize and runtime of TritonPart relative to $hM$ on these benchmarks. In our experiments, we first vary each parameter while keeping the other three constant, and record the cutsize and runtime of TritonPart for the five benchmarks. Next we report the average cutsize and runtime. The results are presented in Figure 5. We normalize these results with respect to the cutsize of TritonPart with default parameter settings. From our findings, we observe that the default setting of the hyperparameters is a reasonable choice.

<table>
<thead>
<tr>
<th>Design</th>
<th>$F_{ avg _ cut }$</th>
<th>$F_{ wst _ cut }$</th>
<th>runtime (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aracne (NG)</td>
<td>0.26</td>
<td>0.72</td>
<td>7.5</td>
</tr>
<tr>
<td>BlackParrot (NG)</td>
<td>0.37</td>
<td>0.59</td>
<td>8.8</td>
</tr>
<tr>
<td>MemPostG-(NG)</td>
<td>0.60</td>
<td>1.01</td>
<td>8.9</td>
</tr>
<tr>
<td>MemPostG-(GG)</td>
<td>0.76</td>
<td>1.09</td>
<td>10.5</td>
</tr>
<tr>
<td>Aracne (GG)</td>
<td>0.73</td>
<td>1.04</td>
<td>10.5</td>
</tr>
<tr>
<td>BlackParrot (GG)</td>
<td>0.83</td>
<td>1.08</td>
<td>11.0</td>
</tr>
</tbody>
</table>

TABLE VI: Results on timing-noncritical paths for $K = 5$ and $\epsilon = 2\%$, $hM$, $KHP$, $TP$, and $T_P$, stand for $hMETIS$, KaHyPar, TritonPart, and timing-driven TritonPart. ['-G': Group, '-C': Cluster, NG: NG45, GF: GF12.]

VI. CONCLUSION AND FUTURE WORK

In this work, we introduce the constraints-driven general hypergraph partitioning problem and present the first open-source constraints-driven general partitioning multi-tool, TritonPart, to tackle it. TritonPart’s adaptation of the multilevel partitioning paradigm, combined with use of efficient algorithms, enables it to effectively manage multiple constraint types such as fixed-vertices, multi-dimensional balance, grouping, embedding, and timing constraints. Extensive experimental results (verifiable using [40]) validate TritonPart’s superior performance on min-cut partitioning, compared to leading partitioners such as SpecPart, $hMETIS$, and KaHyPar. We also demonstrate the effectiveness of TritonPart’s slack propagation-based timing-aware partitioning framework. Our ongoing research pursues three main directions. First, we believe TritonPart’s embedding-aware partitioning could be of independent interest and amenable to machine learning-based applications. In particular, higher-quality embeddings than eigenvectors could potentially guide TritonPart to even better partitioning solutions. Second, we seek to improve the timing-driven framework with use of more accurate delay budget-based methodologies. Third, a spatial arrangement-aware partitioning framework could be beneficial for 3D IC and (multi-)FPGA implementation, where blocks are known to be configured in various arrangements (e.g. stacked on top of each other or in a 2D array).
REFERENCES


