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

Ismail Bustany[†], Grigor Gasparyan[†], 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.gasparyan}@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 [15], [33], [34] and timing-driven partitioning [1]. 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 [2]. 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.[1]
- *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* [37]. 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 multilevel hierarchy of progressively coarser hypergraphs; (ii) *initial partitioning* that partitions the coarsest hypergraph to generate a solution; (iii) *multilevel refinement* that refines the partitioning

[1]We make public with permissive open-source license all results, scripts and code at [40].

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* [17]. 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], *PaToH* [7], *BiPart* [24], *KaHyPar* [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], [16], [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

TABLE I: Terminology and Notation

$w_v \in \mathbb{R}_+^m, w_e \in \mathbb{R}_+^n$	Input weight vectors for vertex v and hyperedge e (input)
K	Number of blocks in output (input)
ϵ	Imbalance parameter for blocks (input)
$V_\gamma(i)$	Set of vertices that must be in block i in the output partition (optional)
$Cmty(v)$	A group of vertices containing vertex v (optional)
$P = \{p_1, p_2, ..., p_l\}$	Set of timing-critical paths (optional)
$P_{non} = \{p_{non_j}\}_{j=1}^{c}$	Set of timing-noncritical paths (optional)
$slack_p$, $slack_e$	Slacks for path p and for hyperedge e (optional)
Δ	Timing delay for a cut hyperedge (optional)
$X \in \mathbb{R}^{\lvert V\rvert \times q}$	Embedding: i^{th} row is the vector of coordinates for vertex i (optional)
$S = \{V_1, V_2, ..., V_K\}$	A partitioning solution $\{V_i \lvert i = 1, 2, ..., K\}$ with K blocks
$Cut(e)$	$Cut(e)$=1 if e is cut by S, 0 otherwise
$I(v)$	Set of the hyperedges incident to vertex v
$block(v)$	The block V_i that vertex v is assigned to
t_p, t_e	Timing weight (cost) for path p and edge e
$D(p)$	The number of times a path p has been cut
$SK(p)$	Normalized slack of a path p
$SF(p)$	Snake factor of a path p
$\alpha \in \mathbb{R}_+^m$	Hyperedge cut cost scaling factor
β, non-negative scalar	Hyperedge timing cost scaling factor
γ, non-negative scalar	Path timing cost scaling factor
τ, non-negative scalar	Snaking cost scaling factor
θ, positive scalar	Number of solutions for cut-overlay clustering and partitioning
η, positive scalar	Number of initial partitions computed after coarsening
η_p, positive scalar	Number of partitions picked from η for V-Cycle refinement
μ, positive scalar	Exponential scaling factor on timing cost for timing-driven partitioning
thr_{ilp}, positive scalar	Maximum number of vertices in a hypergraph allowed for running ILP

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 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 $\mathcal{I}$, we want to compute a partition of V into K disjoint *blocks* $S = \{V_1, \ldots, V_K\}$ that minimizes a cost function:

$$\Phi(\mathcal{I}, S) = \Phi_{cut}(\mathcal{I}, S) + \Phi_{time}(\mathcal{I}, S) \quad (1)$$

where $\Phi_{cut}(\mathcal{I}, S)$ measures the cutsize as in the standard formulation of hypergraph partitioning and $\Phi_{time}(\mathcal{I}, S)$ 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 $v \in V_\gamma(i)$, then vertex v must be in block V_i in the output partition S. One application of this constraint is the preassignment 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:

$$block(v) = block(u) \text{ if } Cmty(v) = Cmty(u) \;\&\&\; Cmty(v) > 1 \quad (2)$$

  Here each community is assigned a unique index greater than 1 and $Cmty(v) = 1$ 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 $1 \leq i \leq K$, $1 \leq j \leq m$.

$$(\frac{1}{K} - \epsilon)W_j \leq \sum_{v \in V_i} w_v(j) \leq (\frac{1}{K} + \epsilon)W_j \quad (3)$$

  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 [32]. 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_{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_{non} that turn critical after partitioning. Essentially, for each timing-noncritical path $p_{non_i} \in P_{non}$, we measure timing-criticality after introducing an extra delay Δ for each time p_{non_i} is cut. If the total delay increment incurred on p_{non_i} exceeds its slack, $slack_{p_{non_i}}$, then we say that p_{non_i} has become timing-critical.

Given the above, and a user-defined parameter α (Table I), we define the cutsize (cut cost) as:

$$\Phi_{cut}(\mathcal{I}, S) = \sum_{e \in E} \langle \alpha, w_e \rangle Cut(e) \quad (4)$$

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 $\Phi_{time}(\mathcal{I}, S)$ 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 *snaking factor* [Section IV-A].

$$\Phi_{time}(\mathcal{I}, S) = \sum_{e=1}^{|E|} \beta t_e Cut(e) + \sum_{p \in \{P\}} (\gamma D(p) t_p + \tau SF(p)) \quad (5)$$

where β, γ, τ 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 [40]. 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 [4] into the multilevel framework. The flow of *TritonPart* is illustrated in Figure 1, with details given in Algorithm 1.

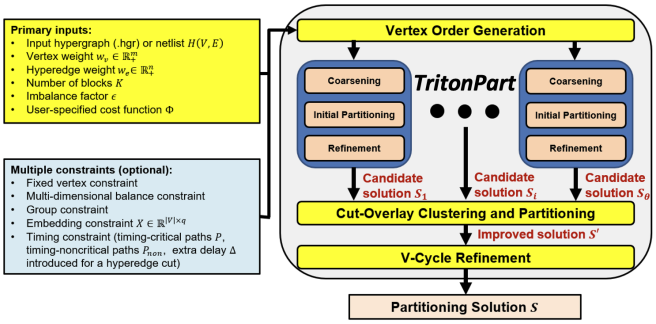


Fig. 1: Flow of the *TritonPart* framework.

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 $\{S_1, ..., S_\theta\}$ where θ 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_c) if the number of vertices in H_c is less than a threshold (thr_{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

Algorithm 1: *TritonPart* framework.

Input: Standard Inputs: H, K, ϵ
Constraint Inputs: $V_\gamma, Cmty, P, P_{non}, slack_p, \Delta, X$
Additional Parameters: θ, η, η_p, thr_{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 $S_{candidate} = \{\}$ for candidate solutions
4 **for** $i = 1$; $i \leq \theta$; $i++$ **do**
  /* Coarsening [Section III-A] */
5 generate random ordering of vertices
6 $\{H_c\} = \{H_{c_1}, H_{c_2}, ..., H_{c_\zeta}\} \leftarrow$ perform constraints-driven First-Choice-based coarsening using the ordering to generate a hierarchy of successively coarser hypergraphs
  /* Initial Partitioning [Section III-B] */
7 $S_{init} \leftarrow$ perform initial partitioning heuristics
8 $S_{init} \leftarrow$ pick the best η_p solutions from S_{init}
  /* Parallel Refinement [Section III-C] */
9 **while** $\{H_c\}.empty() == false$ **do**
10 $\tilde{H}_c \leftarrow \{H_c\}.back()$; $\{H_c\}.pop\_back()$
    /* All the solutions in S_{init} are refined in parallel */
11 **for** *each solution* $S_j \in S_{init}$ **do**
12 perform refinement heuristics on $\tilde{H}_c$ and S_j
13 **end**
14 **end**
15 $S_i \leftarrow$ pick the best solution from S_{init}
16 $S_{candidate}.push\_back(S_i)$
17 **end**
  /* Cut-overlay clustering and partitioning [Section III-D] */
18 $S' \leftarrow$ perform cut-overlay clustering and ILP-based partitioning on $S_{candidate}$
  /* V-Cycle refinement guided by S' [Section III-E] */
19 $\{H_c\} = \{H_{c_1}, H_{c_2}, ..., H_{c_\zeta}\} \leftarrow$ perform constraints-driven First-Choice-based coarsening guided by S' to generate a hierarchy of successively coarser hypergraphs
20 **if** $H_{c_\zeta}.num\_vertices \leq thr_{ilp}$ **then**
21 $S \leftarrow$ perform ILP-based partitioning on H_{c_ζ}
22 **end**
23 **else**
24 $S \leftarrow S'$
25 **end**
26 **while** $\{H_c\}.empty() == false$ **do**
27 $\tilde{H}_c \leftarrow \{H_c\}.back()$; $\{H_c\}.pop\_back()$
28 perform refinement heuristics on $\tilde{H}_c$ and S
29 **end**
30 **return** S

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]:

$$r(u, v) = \sum_{e \in \{I(v) \cap I(u)\}} \frac{\langle \alpha, w_e \rangle}{|e| - 1}. \quad (6)$$

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.

- **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

$$\hat{r}(u,v) = r(u,v) + \rho \frac{1}{||X_u - X_v||_2}, \tag{7}$$

In Equation (7), X_u is the embedding of vertex u, $r(u,v)$ is the score function in Equation (6), and ρ is a normalization factor. We set ρ 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_{coarse}, we define the embedding $X_{v_{coarse}}$ of v_{coarse} to be the center of gravity of the t vertices, i.e. the following convex combination:

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

- **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 [17]. We adapt 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-E).
- **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 η (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_ζ}. We optimize only the cutsize rather than the cost function Φ 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_ζ} does not exceed a threshold thr_{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 then define the following constraints for each $i \in [1,K]$.

- $\sum_{i=1}^{K} x_{v,i} = 1$ for each $v \in V$
- $y_{e,i} \leq x_{v,i}$ for each $e \in E$, and each $v \in e$
- $x_{v,i} = 1$ if $v \in V_\gamma(i)$, i.e., if v is fixed to block V_i
- $W_j \triangleq \sum_{v \in V} w_v(j)$ for each $j \in [1,m]$.
- *Multi-dimensional balance constraint:* for $1 \leq j \leq m$ $(K^{-1} - \epsilon)W_j \leq \sum_{v \in V} w_v(j) x_{v,i} \leq (K^{-1} + \epsilon)W_j$

Observe that the first two constraints enforce these two requirements:

$$x_{v,i} = 1 \text{ iff } v \in V_i \qquad\qquad y_{e,i} = 1 \text{ iff } e \subseteq V_i$$

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

$$max \sum_{1 \leq i \leq K} \sum_{e \in E} \langle \alpha, w_e \rangle y_{e,i}.$$

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_ζ} 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) $\lfloor K/2 \rfloor$ 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 $\lfloor K/2 \rfloor$ 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 $K(K-1)$ 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 [18] 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[2] 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 [4]. Given θ 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 $E_1, \ldots, E_\theta \subset E$. 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 $H_{overlay}$. If the number of vertices in $H_{overlay}$ is less than thr_{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 $H_{overlay}$ and generate a coarser hypergraph $H'_{overlay}$. 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 $H'_{overlay}$. (iii) If the number of vertices in $H'_{overlay}$ is less than thr_{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[3] similar to *hMETIS* [17]. 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_{c_ζ}. In the ILP-based partitioning phase, if the number of vertices in H_{c_ζ} does not exceed thr_{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.

[2]We set the number of passes at each stage of refinement to 2 and the maximum number of moves in each pass to 50.

[3]We set the default iterations of V-Cycle refinement to 2. We stop if the partitioning solution does not improve between successive iterations.

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* [26]. We use the *findPathEnds* function from *Search.hh* available at [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 $slack_p$,

$$t_p = (1 - \frac{slack_p - \Delta}{clock\_period})^\mu \tag{8}$$

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

Snaking factor. The snaking factor [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_0, 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 (5).

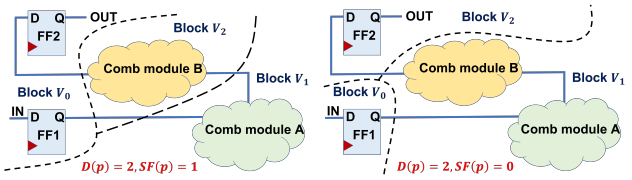
Timing cost for a hyperedge. The timing cost t_e of a hyperedge consists of two parts: (i) the timing weight corresponding to the hyperedge's slack $slack_e$; and (ii) the accumulated timing cost of all paths traversing the hyperedge.

$$t_e = (1 - \frac{slack_e - \Delta}{clock\_period})^\mu + \sum_{\{p|e \in p\}} t_p \tag{9}$$

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 $r_t(u, v)$:

$$r_t(u, v) = \hat{r}(u, v) + \sum_{e \in \{I(v) \cap I(u)\}} \frac{\beta t_e}{|e| - 1} \tag{10}$$



(a) (b)

Fig. 2: Different partitions of timing path p : FF1 $\rightarrow$ Comb module A $\rightarrow$ Comb module B $\rightarrow$ FF2. (a) {FF1, FF2} $\in$ Block V_0, Comb module A $\in$ Block V_1, and Comb module B $\in$ Block V_2. (b) FF1 $\in$ Block V_0, {Comb module A, Comb module B} $\in$ Block V_1, and FF2 $\in$ Block V_2.

C. Timing-aware Refinement

Our timing-aware refinement is based on the cost function in Equation (5).[4] 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 slacks 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.

Line 3: A fixed extra delay Δ (see Section V) is applied to all cut hyperedges in the partitioning solution, i.e., $slack_e = slack_e - \Delta$.

Lines 5-8: The extra delay introduced is propagated by traversing the timing graph. In *TritonPart*, the first vertex v_1 in a hyperedge $e = \{v_1, v_2, ...\}$ is the driver/source vertex and the remaining vertices are load/sink vertices. This convention allows us to interpret the hypergraph as a timing graph. For each timing path that traverses a cut hyperedge e with slack $slack_e$, we first propagate backward, stopping if the slack of the fanout hyperedge is less than $slack_e$. We then propagate forward and stop the propagation if the slack of the fanin hyperedge is less than $slack_e$.

Line 10: The slack of each timing path p is updated based on the minimum slack of all the hyperedges in p, i.e., $slack_p = \min_{e \in p}(slack_e)$.[5]

Line 11: We update the timing cost t_p of all timing paths and the timing cost t_e of all hyperedges, based on Equations (8) and (9).

V. Experimental Setup and Results

TritonPart is implemented using approximately 12K lines of C++ code and is built on the OpenROAD infrastructure [20], [38]. We use *CPLEX* [12] as our default ILP solver but also provide an open-source alternative with an OR-Tools-based implementation [27]. For all reported experimental results, we use *CPLEX* as our primary ILP solver. Given that no prior work has addressed the constraints-driven general hypergraph partitioning problem, there are no existing benchmarks or baselines for direct experimental comparison. We thus divide our validation efforts into: (i) validation of min-cut partitioning (Section V-A), (ii) validation of embedding-aware partitioning (Section V-B), and (iii) validation of timing-driven partitioning (Section V-C). We also present a study on parameter selection in Section V-D. Finally, we explore the effect of multi-starts on *hMETIS* and *TritonPart* in Section V-E.

[4]Recall that our cost function in Equation (5) incorporates cost associated with path cuts $D(p)$ and snaking factor $SF(p)$. This is how we optimize the timing cost tied to $D(p)$ and $SF(p)$.

[5]A path can be interpreted as a sequence of nets (hyperedges) where each net (hyperedge) corresponds to a directed edge in the timing graph that is in the path.

Algorithm 2: Slack propagation.

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

1 $E_{cut} \leftarrow$ identify all hyperedges in H cut by S
2 **for** *each hyperedge* $e \in E_{cut}$ **do**
  /* 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* $p \in P$ **do**
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 of each timing path p in P and the timing cost, t_e of 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 δ = 5, β = 2, γ = 500, ζ = 2, θ = 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, cutsizes 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 (hM_{20}, TP_{20}). Table II and Figure 3 present the results for $K = 2, 3, 4$ respectively with ϵ = 2%, with cutsize values rounded to the nearest integer. In Figure 3 cutsizes are normalized by those obtained from hM_{20}. 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_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_switch* and *bitcoin_miner* benchmarks.

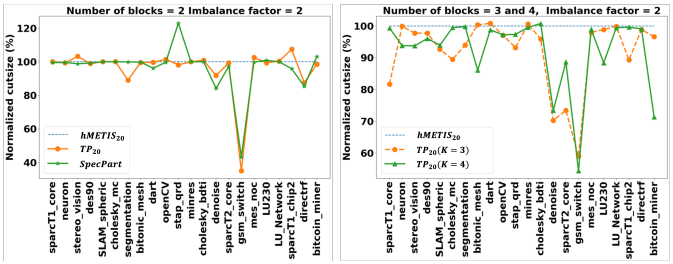


Fig. 3: Results on Titan23 benchmarks for ϵ = 2%. Left: K = 2. Right: K = 3 and 4.

	Statistics		$K=2$			$K=3$		$K=4$	
Benchmark	$\|V\|$	$\|E\|$	hM_{20}	SP	TP_{20}	hM_{20}	TP_{20}	hM_{20}	TP_{20}
sparcT1_core	91976	92827	982	**977**	983	2188	**1787**	2533	**2517**
neuron	92290	125305	245	244	**243**	372	**371**	432	**405**
stereo_vision	94050	127085	171	**169**	176	333	**325**	440	**413**
des90	111221	139557	377	374	**373**	536	**524**	696	**668**
SLAM_spheric	113115	142408	**1061**	**1061**	**1061**	2797	**2592**	3371	**3167**
cholesky_mc	113250	144948	**282**	**282**	**282**	886	**793**	983	**978**
segmentation	138295	179051	120	120	**107**	476	**447**	496	**495**
bitonic_mesh	192064	235328	585	584	**582**	**895**	898	1304	**1123**
dart	202354	223301	837	**805**	834	**1190**	1200	1430	**1413**
openCV	217453	284108	435	**434**	442	502	**487**	526	**512**
stap_qrd	240240	290123	377	464	**370**	501	**467**	715	**696**
minres	261359	320540	**207**	**207**	**207**	**309**	311	407	**405**
cholesky_bdti	266422	342688	**1156**	**1156**	1166	1769	**1697**	**1874**	1889
denoise	275638	356848	497	**418**	456	953	**670**	1172	**860**
sparcT2_core	300109	302663	1221	**1188**	1214	2827	**2076**	3324	**2948**
gsm_switch	493260	507821	4235	1833	**1483**	4149	**2446**	5169	**2813**
mes_noc	547544	577664	635	**633**	651	1164	**1140**	1315	**1302**
LU230	574372	669477	3334	3363	**3314**	4550	**4498**	6325	**5591**
LU_Network	635456	726999	**524**	**524**	526	787	**786**	1496	**1488**
sparcT1_chip2	820886	821274	914	**876**	982	1453	**1298**	1610	**1604**
directrf	931275	1374742	603	**515**	527	728	**720**	1104	**1093**
bitcoin_miner	1089284	1448151	1514	1562	**1492**	1945	**1879**	2605	**1860**

TABLE II: Cutsize comparisons for *hMETIS*, *SpecPart*, and *TritonPart* on Titan23 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*, *sparcT1_chip2* and *directrf* benchmarks. These embeddings consist of the first two nontrivial eigenvectors of a generalized eigenvalue problem [4]. 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_emb_{20} represents the best cutsize across twenty runs. TP_emb_{20} produces partitioning solutions that outperform those of *hMETIS*, *SpecPart*, and *TritonPart* without embedding constraints by $\sim$11%, $\sim$2%, and $\sim$8%, 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*).

	cutsize					***runtime (second)***				
Benchmark	hM_{20}	SP	TP_{20}	TP_emb	TP_emb_{20}	hM_{20}	SP	TP_{20}	TP_emb	TP_emb_{20}
dart	837	805	835	804	**784**	444	93	1360	46	920
denoise	497	418	454	418	**416**	696	78	1800	88	1760
sparcT1_chip2	914	**876**	997	882	877	1800	292	5400	306	6120
directrf	603	515	526	657	**493**	1883	378	3260	203	4060

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 I-A, there are no open-source timing-driven partitioners available for direct comparison. Toward a fair comparison, we propose reasonable baselines based on *hMETIS* and *KaHyPar*, illustrated in Figure 4. Our evaluation flow comprises the following steps. (i) We extract the top (non-decreasing order of slack) 100,000 timing paths in the netlist using OpenSTA [26]. (ii) Hyperedge weights are updated based on the slack values of corresponding nets according to Equation (9). (iii) The reweighted hypergraph is passed as input to *TritonPart*, *hMETIS*, and *KaHyPar*. (iv) We evaluate the partitioning solutions from *TritonPart*, *hMETIS*, *KaHyPar*, and timing-driven *TritonPart* (***TritonPart***$_t$) and report the timing metrics.

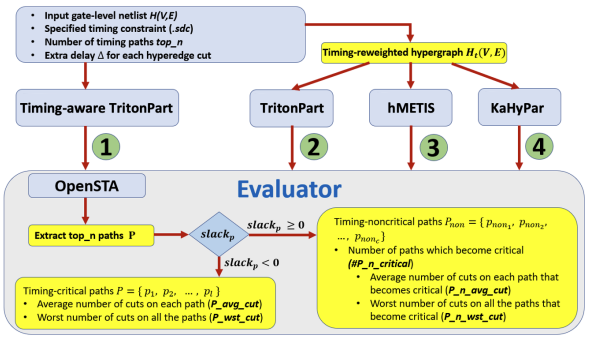


Fig. 4: Timing-driven partitioning evaluation flow.

Benchmark	$\|V\|$	$\|E\|$	***cp(ns)***	***Δ (ns)***	***#paths***	***#c_paths***	***#nc_paths***
Ariane (NG45)	118061	121289	1.0	0.2	44428	16127	28301
BlackParrot (NG45)	768986	808973	1.3	0.26	100000	39872	60128
MemPool Group (NG45)	2741149	2788222	3.8	0.76	100000	15083	84917
MemPool Cluster (NG45)	10486897	10726018	4.6	0.92	100000	38720	61280
Ariane (GF12)	9947	100620	–	20%	43963	15184	28779
BlackParrot (GF12)	686364	700967	–	20%	100000	18909	81091
MemPool Group (GF12)	2460278	2488257	–	20%	100000	21111	78889

TABLE IV: Benchmarks and Statistics. Δ is normalized to clock period (*cp*) for GF12. *#c_paths* and *#nc_paths* denote numbers of time-critical and noncritical paths respectively.

In our experimental setup, for each design, we set Δ to be 20% of the design's clock period. We use modern VLSI benchmarks from the *MacroPlacement repository* [37], implemented in open NanGate45 (NG45) and commercial GlobalFoundries 12nm (GF12) enablements. Benchmark statistics are given in Table IV.

Evaluation of cuts on timing-critical paths. Results for cuts on timing-critical paths are detailed in Table V. For the majority of the benchmarks, timing-driven *TritonPart* (TP_t) outperforms other baselines in both *P_avg_cut* and *P_wst_cut* metrics, albeit with some runtime overhead. For the *MemPool Cluster* benchmark which has more than 10M vertices, both *hMETIS* and *KaHyPar* crash, while *TritonPart* successfully runs to completion without cutting any timing-critical paths. Similarly, for the large *MemPool Group* design with GF12 enablement, TP_t does not cut any timing-critical paths, in contrast to *hMETIS*, *KaHyPar*, and *TritonPart*. These results validate the timing-driven capabilities of *TritonPart*, and its ability to handle very large problem instances.

Evaluation of cuts on timing-noncritical paths. Results for cuts on timing-noncritical paths are presented in Table VI. Across all benchmarks, TP_t outperforms other baselines in all metrics (*#P_n_critical*, *P_n_avg_cut*, and *P_n_wst_cut*). Notably, for the *MemPool Group* design with GF12 enablement, TP_t achieves a $\sim$21X, $\sim$119X and $\sim$66X reduction in *#P_n_critical* compared to *hMETIS*, *KaHyPar* and *TritonPart*, respectively. For the same design, TP_t achieves a $\sim$1.8X, $\sim$2X and $\sim$1.3X reduction in *P_n_avg_cut* compared to *hMETIS*, *KaHyPar* and *TritonPart* respectively. For the *P_n_wst_cut* metric on the *BlackParrot* design with NG45 enablement, TP_t achieves 5X, 3X and 3X reduction compared to *hMETIS*, *KaHyPar* and *TritonPart*.

These results suggest that the timing-driven *TritonPart* can help avoid timing-noncritical paths becoming critical.

	P_avg_cut				*P_wst_cut*				*runtime (min)*			
Design	***hM***	***KHPr***	***TP***	***TP$_t$***	***hM***	***KHPr***	***TP***	***TP$_t$***	***hM***	***KHPr***	***TP***	***TP$_t$***
Ariane (NG)	0.61	0.63	0.44	**0.12**	7	6	6	**3**	0.3	1	2	6
BlackParrot (NG)	**0.00**	0.37	0.46	0.26	**0**	2	3	1	2	117	9	16
MemPool-G (NG)	0.08	0.00	0.00	0.00	2	0	4	0	15	417	24	28
MemPool-C (NG)	–	–	0.00	0.00	–	–	0	0	–	–	77	84
Ariane (GF)	0.09	0.09	0.32	**0.02**	4	4	5	**2**	0.3	1	4	10
BlackParrot (GF)	0.13	0.29	0.66	**0.06**	1	1	3	1	3	108	13	25
MemPool-G (GF)	0.03	0.21	0.19	**0.00**	2	5	2	**0**	12	258	26	37

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

	#P_n_critical				*P_n_avg_cut*				*P_n_wst_cut*			
Design	***hM***	***KHPr***	***TP***	***TP$_t$***	***hM***	***KHPr***	***TP***	***TP$_t$***	***hM***	***KHPr***	***TP***	***TP$_t$***
Ariane (NG)	442	571	569	**162**	2.92	2.79	1.59	**1.05**	4	4	4	**3**
BlackParrot (NG)	8407	6756	9011	**3128**	1.32	1.12	1.15	**1.00**	5	3	3	**1**
MemPool-G (NG)	3770	0	6244	0	1.13	0.00	2.02	0.00	2	0	4	0
MemPool-C (NG)	–	–	0	0	–	–	0.00	0.00	–	–	0	0
Ariane (GF)	0	0	1454	0	0.00	1.01	1.00	0.00	0	0	2	**0**
BlackParrot (GF)	13954	15511	21752	**786**	1.38	1.09	1.40	**1.01**	4	4	4	**2**
MemPool-G (GF)	2084	11553	6441	**97**	1.86	2.01	1.32	**1.00**	2	6	2	**1**

TABLE VI: Results on timing-noncritical paths for $K = 5$ and $\epsilon = 2\%$. hM, $KHPr$, TP, and TP_t 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) ***thr_coarsen_hyperedge_size_skip***: hyperedges of size larger than this threshold are excluded from the constraints-driven coarsening phase. (ii) ***coarsening_ratio***: the maximum ratio between the numbers of vertices of two successive hypergraphs in the multilevel hierarchy. (iii) ***max_moves***: the maximum number of vertices that can be moved in each pass of the refinement phase. (iv) ***num_coarsen_solution***: the number of candidate solutions $S_{candidate}$, each 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*, *LU_Network*, *sparcT1_chip2*, *directrf* and *bitcoin_miner*. We define the score value as the average improvement in cutsize and runtime of *TritonPart* relative to hM_{20} 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.

E. QoR vs. Runtime Comparison

Leading multilevel partitioners, such as *hMETIS*, often employ multi-start strategies to improve their performance. As *TritonPart* follows a similar approach, we analyze the effect of multi-starts on both *hMETIS* and *TritonPart*. Our methodology consists of the following steps. (i) We generate ψ ($1 \leq \psi \leq 50$) solutions using both *hMETIS* and *TritonPart*, varying the random seed for each trial. (ii) We record the best solution for each ψ. The result is presented in Figure 6 for the *bitcoin_miner* benchmark with $K = 4$ and $\epsilon = 2\%$. Unsurprisingly, with additional multi-starts both *hMETIS* and *TritonPart* can generate better cutsize. However, *TritonPart* consistently outperforms *hMETIS*, achieving ∼30% cutsize reduction with a ∼2.4X runtime overhead. More crucially, *TritonPart* demonstrates better robustness and stability across random seeds compared to *hMETIS*.

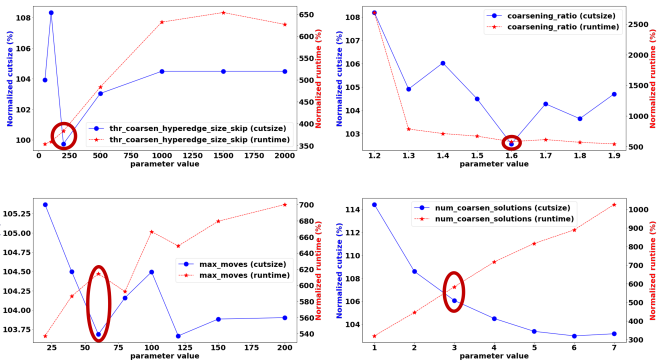


Fig. 5: Validation of *TritonPart* parameters.

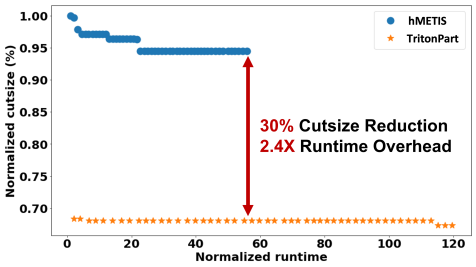


Fig. 6: Cutsize versus runtime comparison of *hMETIS* and *TritonPart* on the *bitcoin_miner* benchmark for $K = 4$ and $\epsilon = 2\%$.

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).

Acknowledgments. This work was partially supported by DARPA HR0011-18-2-0032, and by NSF grants CCF-2112665, CCF-2039863 and CCF-1813374.

REFERENCES

[1] C. Ababei, S. Navaratnasothie, K. Bazargan and G. Karypis, "Multi-objective circuit partitioning for cutsize and path-based delay minimization", *Proc. ICCAD*, 2002, pp. 181-185.

[2] D. Avdiukhin, S. Pupyrev and G. Yaroslavtsev, "Multi-dimensional balanced graph partitioning via projected gradient descent", *Proc. VLDB Endowment* 12(8) 2019, pp. 906-919.

[3] R. Burra and D. Bhatia, "Timing driven multi-FPGA board partitioning", *Proc. Intl. Conf. on VLSI Design*, 1998, pp. 234-237.

[4] I. Bustany, A. B. Kahng, I. Koutis, B. Pramanik and Z. Wang, "SpecPart: A supervised spectral framework for hypergraph partitioning solution improvement", *Proc. ICCAD*, 2022, pp. 1-9.

[5] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved algorithms for hypergraph bipartitioning", *Proc. ASP-DAC*, 2000, pp. 661-666.

[6] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Iterative partitioning with varying node weights", *VLSI Design* 11(3) 2000, pp. 249-258.

[7] Ü. Çatalyürek and C. Aykanat, "PaToH (partitioning tool for hypergraphs)", Boston, MA, Springer US, 2011.

[8] J. Cong and C. Wu, "Global clustering-based performance-driven circuit partitioning", *Proc. ISPD*, 2002, pp. 149-154.

[9] J. Cong and S. K. Lim, "Multiway partitioning with pairwise movement", *Proc. ICCAD*, 1998, pp. 512-516.

[10] J. Cong, S.K. Lim and C. Wu, "Performance driven multi-level and multiway partitioning with retiming", *Proc. DAC*, 2000, pp. 274-279.

[11] G. Karypis, "Multilevel hypergraph partitioning", Boston, MA, Springer US, 2003.

[12] IBM ILOG CPLEX optimizer, https://www.ibm.com/analytics/cplex-optimizer.

[13] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions", *Proc. DAC*, 1982, pp. 175-181.

[14] G. Karypis and V. Kumar, "hMETIS, a hypergraph partitioning package, version 1.5.3", 1998. http://glaros.dtc.umn.edu/gkhome/fetch/sw/hMETIS/manual.pdf

[15] D. J.-H. Huang and A. B. Kahng, "Multi-way system partitioning into a single type or multiple types of FPGAs", *Proc. FPGA*, 1995, pp. 140-145.

[16] W. Hou, X. Hong, W. Wu and Y. Cai, "A path-based timing-driven quadratic placement algorithm", *Proc. ASP-DAC*, 2003, pp. 745-748.

[17] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning", *Proc. DAC*, 1999, pp. 343-348.

[18] G. Karypis, R. Aggarwal, V. Kumar and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI domain", *IEEE Trans. on VLSI* 7(1) 1999, pp. 69-79.

[19] A. B. Kahng and X. Xu, "Local unidirectional bias for smooth cutsize-delay tradeoff in performance-driven bipartitioning", *Proc. ISPD*, 2003, pp. 81-86.

[20] A. B. Kahng and T. Spyrou, "The OpenROAD project: unleashing hardware innovation", *Proc. GOMACTech*, 2021.

[21] T. Luo, D. Newmark and D. Z. Pan, "A new LP based incremental timing driven placement for high performance designs", *Proc. DAC*, 2006, pp. 1115-1120.

[22] J. Minami, T. Koide and S. Wakabayashi, "A circuit partitioning algorithm under path delay constraints", *Proc. IEEE APCCAS*, 1998, pp. 113-116.

[23] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, et al., "A graph placement methodology for fast chip design", *Nature*, 594(7862) 2021, pp. 207-212.

[24] S. Maleki, U. Agarwal, M. Burtscher and K. Pingali, "BiPart: a parallel and deterministic hypergraph partitioner", *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2021, pp. 161-174.

[25] K. E. Murray, S. Whitty, S. Liu, J. Luu and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD", *Proc. Intl. Conf. on Field programmable Logic and Applications*, 2013, pp. 1-8.

[26] OpenSTA static timing analyzer, https://github.com/The-OpenROAD-Project/OpenSTA

[27] Google OR-Tools version 9.4, https://developers.google.com/optimization/

[28] S. Ou and M. Pedram, "Timing-driven bipartitioning with replication using iterative quadratic programming", *Proc. ASP-DAC*, 1999, pp. 105-108.

[29] D. Z. Pan, B. Halpin and H. Ren, "Timing-driven placement", *Handbook of Algorithms for Physical Design Automation*, 2008, pp. 423-446.

[30] L. A. Sanchis, "Multiple-way network partitioning with different cost functions", *IEEE Trans. on Computers* 42(12) 1993, pp. 1500-1504.

[31] S. Schlag, T. Heuer, L. Gottesbüren, Y. Akhremtsev, C. Schulz and P. Sanders, "High-Quality Hypergraph Partitioning", *ACM J. Exp. Algorithmics*, 27(1.9) 2023, pp. 1-39

[32] J. Sybrandt, R. Shaydulin and I. Safro, "Hypergraph partitioning with embeddings", *IEEE Trans. on Knowledge and Data Engineering* 34(6) 2022, pp. 2771-2782.

[33] D. Zheng, X. Zang and M. D.F. Wong, "TopoPart: a multi-level topology-driven partitioning framework for multi-FPGA systems", *Proc. ICCAD*, 2021.

[34] D. Zheng and E. F. Y. Young, "An integrated circuit partitioning and TDM assignment optimization framework for multi-FPGA systems", *Proc. ASP-DAC*, 2023, pp. 522-526.

[35] Combinational logic snake paths. http://www.truevue.org/p/847

[36] Hypergraph partitioning for VLSI: benchmarks, code and leaderboard, https://github.com/TILOS-AI-Institute/HypergraphPartitioning.

[37] C.-K. Cheng, A. B. Kahng, S. Kundu, Y. Wang and Z. Wang, "Assessment of reinforcement learning for macro placement", *Proc. ISPD*, 2023, pp. 158-166. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/CodeElements/Grouping.

[38] The OpenROAD project. https://github.com/The-OpenROAD-Project/OpenROAD

[39] Search.hh. https://github.com/The-OpenROAD-Project/OpenSTA/blob/master/include/sta/Search.hh

[40] TritonPart GitHub repository for *TritonPart*. https://github.com/ABKGroup/TritonPart_OpenROAD