ChipletPart: Cost-Aware Partitioning for 2.5D Systems

ALEXANDER GRAENING and PUNEET GUPTA, University of California, Los Angeles, USA
ANDREW B. KAHNG, BODHISATTA PRAMANIK, and ZHIANG WANG, University of California, San Diego, USA

Industry adoption of chiplets has been growing as chiplets are a cost-effective option for making large, high-performance systems. Consequently, partitioning large systems into chiplets is increasingly important. In this work, we introduce *ChipletPart* — a cost-driven 2.5D system partitioner that addresses the unique constraints of chiplet systems, including complex objective functions, limited reach of inter-chiplet I/O transceivers, and the assignment of heterogeneous manufacturing technologies to different chiplets. *ChipletPart* integrates a sophisticated chiplet cost model with a genetic algorithm (GA)-based technology assignment and partitioning methodology, along with a simulated annealing (SA)-based chiplet floorplanner. Our results show that *ChipletPart*: (i) reduces chiplet cost by up to 58% (20% geometric mean) compared to state-of-the-art min-cut partitioners, which often yield floorplan-infeasible solutions; (ii) generates partitions with up to 47% (6% geometric mean) lower cost compared to the prior work *Floorplet*; (iii) reduces chiplet cost up to 48% (30% geometric mean) compared to *Chipletizer*, while consistently producing I/O-feasible chiplet solutions across all testcases; and (iv) for the testcases we study, heterogeneous integration reduces cost by up to 43% (15% geometric mean) compared to homogeneous implementations. Additionally, we explore Bayesian optimization (BO) for finding low cost and floorplan-feasible chiplet solutions with technology assignments. On some testcases, our BO framework achieves better system cost (up to 5.3% improvement) with higher runtime overhead (up to 4×) compared to our GA-based framework. We also present case studies that show how changes in packaging and inter-chiplet signaling technologies can affect partitioning solutions. Finally, *ChipletPart*, the underlying chiplet cost model, and our chiplet testcase generator are available as open-source tools for the community.

CCS Concepts: • **Hardware → 3D integrated circuits**; **Partitioning and floorplanning**; **Economics of chip design and manufacturing**.

Additional Key Words and Phrases: 2.5D, Chiplets, Partitioning, Floorplanning, Chiplet Manufacturing Cost Model

1 INTRODUCTION

The integration of multiple chips on an interposer has become a favorable approach to reduce the cost of building large systems [1, 2]. In a 2.5D system, a design is decomposed into multiple smaller chiplets, which are packaged together on the same substrate. This splitting can have significant benefits for yield and can enable designs using heterogeneous process technology nodes, going beyond what is possible in a monolithic design. On the other hand, inter-die communication exhibits higher area and power overhead compared to intra-die communication. Consequently, the potential cost benefits of disaggregation are not always realized [3]. Therefore, it is important to intelligently partition a design into constituent chiplets *and* assign a manufacturing technology to each chiplet so as to optimize manufacturing costs.

The natural partitioning granularity for 2.5D integration is at the block-level. Splitting individual IP blocks into multiple chiplets is problematic for design and test methodology reasons as well as for IP reuse. Additionally, splitting an IP block will often have a significant impact on performance and I/O count. Due to these considerations, block-level chiplet partitioning (Figure 1) is the focus of this paper. The cost-aware partitioning problem for chiplet systems is fundamentally different from the well-studied netlist min-cut partitioning [4–6]. For the former, the problem size is smaller (usually a few hundred blocks and a few tens of chiplets at most) and the underlying objective function can

Authors' addresses: Alexander Graening, agraening@ucla.edu; Puneet Gupta, puneetg@ucla.edu, University of California, Los Angeles, Los Angeles, California, USA; Andrew B. Kahng, abk@ucsd.edu; Bodhisatta Pramanik, bopramanik@ucsd.edu; Zhiang Wang, zhw033@ucsd.edu, University of California, San Diego, La Jolla, California, USA.

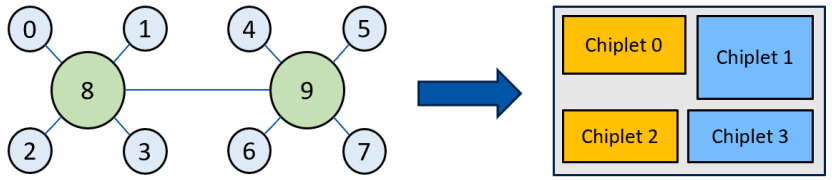

Fig. 1. Generic example of block-level chiplet partitioning. Left: an IP block-level netlist (different types of IP blocks shown in different colors). Right: an integrated 2.5D chiplet system reflecting the block-level netlist partitioning with technology node assignment shown in different colors.

be complex. Furthermore, inter-chiplet I/O reach limitations can render some partitions infeasible, necessitating floorplan-awareness in the partitioner. This is further complicated by the need to select a manufacturing technology for each chiplet.

We propose *ChipletPart*, the first open-source, unified framework for chiplet partitioning that combines *cost-driven* multi-way partitioning, heterogeneous technology assignment and I/O *reach-aware* floorplanning in a single, automated flow. *ChipletPart* presents a novel adaptation of classical optimizers such as genetic algorithms (GA) and simulated annealing (SA) to generate floorplan-feasible, cost-optimized 2.5D chiplet system partitions with technology assignments. *ChipletPart* is intended for design space exploration before physical implementation.[1] Our key contributions include:

- **Integrated chiplet partitioning with floorplan-feasibility:** We present the first unified framework that optimizes system partitioning while guaranteeing feasibility under I/O reach constraints. In contrast to standard min-cut partitioners (e.g., *hMETIS* [4], *TritonPart* [6]) that ignore floorplan-feasibility, *ChipletPart* employs a reach-aware, simulated annealing-based placement engine to guarantee floorplan-feasible solutions (Section 5.2).
- **Heterogeneous technology assignment:** We handle technology node assignment during partitioning using a genetic algorithm. This enables exploration of cost trade-offs in heterogeneous 2.5D systems — a capability not supported by existing tools such as *Floorplet* [7] or *Chipletizer* [8]. Our heterogeneous technology-aware, cost-driven, multi-way partitioner, *ChipletPart*, finds cost reductions of up to 43% (6% geometric mean) in multi-technology scenarios. To the best of our knowledge, we are the first work that does technology assignment during partitioning (Section 3). In addition, we explore the use of Bayesian optimization (BO) to search for floorplan-feasible, cost-*optimal* chiplet solutions with technology assignments. While BO improves system cost by up to 5.3% over our GA-based framework on a few testcases, it incurs a runtime overhead of up to 4×. Thus, GA and BO provide a quality-runtime trade-off (Section 6.4).
- **Improvements over SOTA:** *ChipletPart* achieves up to 46% improvement in cost over state-of-the-art min-cut based partitioners (*hMETIS* [4], *TritonPart* [6]) that do not guarantee floorplan-feasibility. Compared to the *parChiplet* partitioner from *Floorplet* [7], *ChipletPart* generates chiplet solutions that are up to 47% (6% geometric mean) better in cost. Compared to manual partitions, *ChipletPart* generates 34% (13% geometric mean) better solutions (Section 6). We also compare against the cost-driven *Chipletizer* [8] framework and observe that *ChipletPart* consistently produces I/O-feasible partitions with up to 48% lower cost across our design suite (Section 6). Additionally, we explore how variations in technology parameters affect chiplet partitioning (Section 7).

[1] *ChipletPart* considers a given architecture and generates partitioned chiplets before physical implementation. We do not perform architecture exploration in this work. We envision *ChipletPart* as a tool to assist human designers by generating strong initial solutions, which can then be refined using domain expertise.

- **Standardized open-source ecosystem:** We translated the Python-based cost model [9, 10] to an equivalent C++ implementation (Section 4.1). This yields a 5× speedup in cost model execution runtime, and a speedup of more than 100× in our multithreaded implementation. *ChipletPart*, along with its core cost model, is permissively open-sourced [11], enabling others to readily adapt it for benchmarking and further development. Also, we make our testcases publicly available, enabling the community to access a new, standardized set of benchmarks.

In the following, Section 2 reviews related works on chiplet cost modeling, partitioning and floorplanning. Sections 3 and 5 respectively provide an overview and details of our partitioning approach. Section 4 describes the driving considerations in chiplet partitioning. Sections 6 and 7 show experimental results and additional case studies, and Section 8 concludes the paper.

2 RELATED WORK

We now discuss fundamentals and previous works on chiplet cost modeling, then review existing works on chiplet partitioning and floorplanning. Tables 1 and 2 summarize key terms and notations.

2.1 Chiplet Cost Modeling

Cost reduction is one of the main drivers for developing 2.5D systems. The total cost of developing a VLSI system can be divided into two parts: non-recurring engineering (NRE) cost and recurring engineering (RE) cost [12]. NRE cost refers to the one-time cost of designing a VLSI system, including IP qualification, architecture simulation, verification, physical design, software license fees, etc. RE cost refers to the fabrication costs in mass production, such as wafers, assembly, and test. Researchers have proposed several chiplet cost models to estimate various components of the total 2.5D system cost. [12] introduces a quantitative cost model for comparing RE and NRE costs between monolithic SoC and multi-chip integration, but accounts for fewer RE cost considerations than our chosen model. [13] and [7] propose cost models that take into account reliability issues such as bump stress and warpage. [3, 9] present a case study of a large system built using chiplets and analyze the sensitivity of system cost to factors such as defect density, assembly cost, I/O size, etc. The authors have made their cost model publicly available at [10]. In this work, we use the cost model proposed by [10] (see Section 4) due to the wide range of configurable parameters available for system technology co-optimization (STCO).[2] The cost model in [9] was used in collaboration with IMEC to perform an STCO study in [14]. While the specific parameter settings in the cost model do influence partitioning outcomes, all parameters are externally configurable via user-defined configuration files. To enable efficient integration with our partitioning framework, we ported the cost model to C++, resulting in significantly improved performance.

2.2 Chiplet Partitioning and Floorplanning

Several previous works address chiplet partitioning. [15] incorporates the min-cut partitioner *hMETIS* [16] within a chiplet implementation flow. [7] proposes *parChiplet*, which partitions the SoC system into chiplets based on functional and area characteristics of IP blocks. However, [7, 15] do not optimize the actual chiplet cost. *Chipletizer* [8] proposes a unified design characterization graph to represent both SoC designs and chiplets, and uses simulated annealing (SA) to optimize the overall cost of chiplet-based systems. However, *Chipletizer* does not consider floorplan feasibility or support heterogeneous integration. The recent work of [17] uses reinforcement learning (RL) and simulated annealing to perform PPA-oriented chiplet partitioning. However, their method does not guarantee

[2]To the best of our knowledge, the cost model in [10] is the most detailed open-source cost model currently available for chiplets at the time of this publication.

Table 1. General terminology and notation.

Notation	Description
C	Set of chiplets
C_t	Set of chiplets implemented in technology node t
N	Chiplet-level netlist
$\mathcal{S}$	Block-level netlist
V	Chip manufacturing volume
$\mathcal{T}$	Set of technology nodes
ω	Mapping from chiplets to technology nodes
ϕ	Partitioning objective function
m	Number of different technology nodes

Table 2. GA terminology and notation. Default hyperparameters were empirically determined (Section 6.2.5).

Notation	Description
tot_{pop}	Number of members in the population (default 50)
k_{pop}	Number of pairs of parents selected from a population for tournament selection (default 45)
ζ	Tournament size (default 3)
σ	Number of members picked for elitism (default 5)
$\Delta_{threshold}$	Threshold value for improvement between successive generations (default 0.01)
Ψ	Maximum number of generations (default 50)
ϵ	Maximum number of successive generations with cost improvement less than $\Delta_{threshold}$ (default 10)
p_c	Crossover probability (default 0.60)
p_m	Mutation probability (default 0.07)
K_{max}	Maximum number of attainable chiplets (default 8)

Table 3. Comparison of state-of-the-art chiplet partitioning methods.

Methods	Cost-Driven	Floorplan Feasibility	Heterogeneous Integration
[7], [15], [16]			
[8]	✓		
[17]	✓		✓
ChipletPart	✓	✓	✓

floorplan-feasibility. The main differences between our *ChipletPart* and previous works are summarized in Table 3. In our experimental evaluations, we compare *ChipletPart* with [7] and [16]. The authors of [8, 15, 17] have not released their source code or binaries.[3]

Existing chiplet floorplanning approaches fall into three categories: simulated annealing-based, branch-and-bound (B&B)-based and mathematical programming (MP)-based. Works such as [18, 19] use classical floorplan representations such as B* tree and apply SA to optimize an objective function. [20, 21] enumerate possible floorplanning solutions and apply B&B to find a near-optimal

[3]We attempted to contact the authors of [17], but have not received a response.

solution. [7, 13] develop MP-based formulations of chiplet floorplanning, enabling MP solvers to find optimal solutions. We note that existing works fail to handle the "reach" constraints imposed by I/O cells, and generally assume that the size and shape of chiplets are predetermined by designers. In this work, we propose a reach-aware chiplet floorplanner that simultaneously determines the location and shape for each chiplet.

3 OUR APPROACH

We now introduce the chiplet partitioning problem, then give an overview of our chiplet partitioning framework.

3.1 Problem Formulation

The chiplet partitioning problem that we address is fundamentally different from classical min-cut partitioning. The key differences are summarized as follows.

- **Problem size**: Min-cut partitioning is typically performed on gate-level netlists comprising millions of gates. By contrast, we focus on a block-level netlist that typically comprises several hundreds of IP blocks.
- **Objective:** Min-cut partitioning focuses on minimizing *cutsize*, the number of nets (hyperedges) crossing partition boundaries. By contrast, we seek to reduce 2.5D system development cost, which brings a more complex set of considerations (Section 4).
- **Constraints:** Min-cut partitioning minimizes cutsize subject to a given balance constraint [6]. By contrast, chiplet partitioning is not necessarily balanced, but it must produce *I/O-feasible* solutions (Section 5.2).
- **Heterogeneity:** Min-cut partitioning is insensitive to partition labels: swapping the contents of two partitions will not affect cutsize. By contrast, chiplet partitioning is sensitive to partition labels (i.e., technology node assignments): swapping the technologies of two chiplets can significantly change total cost of a 2.5D system.

These fundamental differences between min-cut partitioning and chiplet partitioning motivate our studies. Formally, the inputs to our multi-technology chiplet partitioning framework are a block-level netlist $\mathcal{S}$ and a set of technology nodes $\mathcal{T}$. The outputs are a set of chiplets C and their corresponding technology assignment $\omega : C \rightarrow \mathcal{T}$. The objective is to minimize the total cost of a 2.5D system:

$$\phi_{overall}(C) = \frac{\phi_{assembly} + \sum \frac{\phi_{die}}{Y_{die}}}{Y_{assembly}} + \frac{g(\omega)}{V} \tag{1}$$

where ϕ_{die} is the silicon cost of each chiplet, $\phi_{assembly}$ is the assembly cost, Y_{die} is the die yield, $Y_{assembly}$ is the assembly yield, $g(\omega)$ is the non-recurring engineering (NRE) cost, and V is the chip manufacturing volume. The constraint is the I/O-feasibility of the chiplet set C.

3.2 Overview of *ChipletPart* Framework

We now describe our chiplet partitioning framework, *ChipletPart* (see Figure 2). Our *ChipletPart* differs from classical min-cut partitioners [4–6] and previous approaches [2, 7, 8, 15] in several key respects. (i) We optimize a comprehensive chiplet cost function (Section 4), unlike min-cut partitioners that focus on minimizing net cutsize. (ii) We integrate a fast, Go-With-the-Winners [22] SA-based chiplet floorplanner to ensure I/O-feasibility of a partitioning solution (Section 5.2). (iii) We leverage a genetic algorithm (GA) framework to discover high-quality technology node assignments for heterogeneous integration.

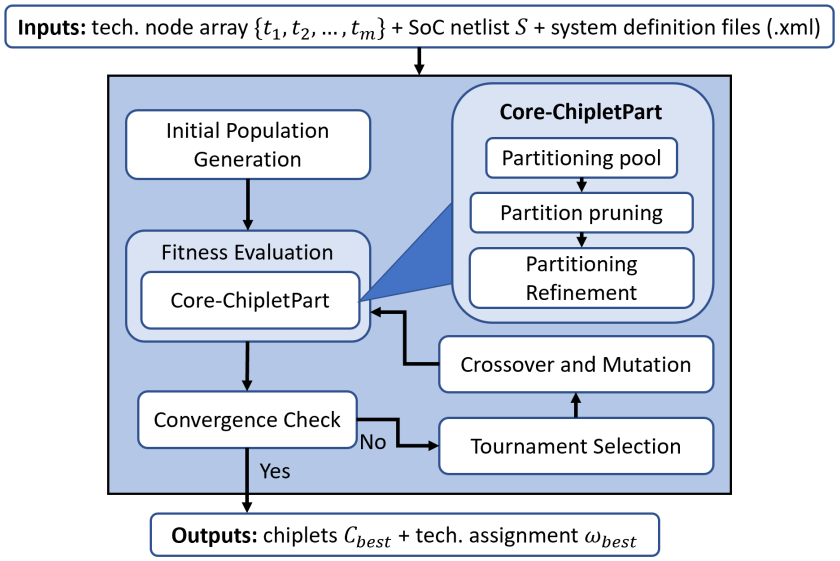

Fig. 2. *ChipletPart* Framework. *Core-ChipletPart* is shown in Figure 5. Partitioning refinement is shown in Figure 6.

ChipletPart inputs include a system block-level netlist $\mathcal{S}$, a set of technology nodes $\mathcal{T}$, and system definition provided in XML files.[4] The algorithm is outlined in Algorithm 1 and visually illustrated in Figure 3. Within the GA framework, a *gene* is a technology node, and a *genome* is a sequence of technology nodes in which corresponding chiplets (partitions of $\mathcal{S}$) will be implemented. Details and source code for our GA-based partitioning and technology assignment methodology are in [11].

Step 1: [Lines 3-4] We first generate the initial population in the GA. Each member in the population (a *genome*) represents an ordered sequence of technology assignments (to chiplets) $\omega : \mathcal{C} \rightarrow \mathcal{T}$. We generate tot_{pop} (tot_{pop} = 50 by default) genomes; each genome is generated by randomly mapping $K_{\max}$ chiplets to technology nodes $\mathcal{T}$.[5] To avoid redundant evaluations and improve scalability, we canonicalize all generated genomes by mapping functionally equivalent assignments to a unique representation. For example, genomes $\langle 7\,\text{nm}, 7\,\text{nm}, 14\,\text{nm}\rangle$ and $\langle 14\,\text{nm}, 7\,\text{nm}, 7\,\text{nm}\rangle$ are treated as equivalent, since they correspond to the same multiset of assignments.

Step 2: [Lines 5-11] We assess the fitness of each genome in the current population. For each genome ω_j, we (i) run *Core-ChipletPart*[6] to generate a partitioning solution based on ω_j and (ii) calculate the *fitness score* of this solution using our cost model.

Step 3: [Lines 12-21] We declare convergence if either of the following two termination criteria is met:

- The algorithm reaches the maximum number of generations, Ψ (Ψ = 50 by default). Upon reaching this limit, the algorithm terminates and returns the best chiplet solution, C_{best}, along with its corresponding technology mapping, ω_{best}.

[4] As in [3, 9], our netlists are modeled as graphs rather than hypergraphs, since these previous works use a directed (source-sink) edge for every inter-block connection.

[5] For simplicity, we set $tot_{pop} = k_{pop} + \sigma$.

[6] For better scalability, we run *Core-ChipletPart* with fewer FM (Fiduccia-Mattheyses) moves and passes, and fewer *initial* solutions. Details are seen in our code [11].

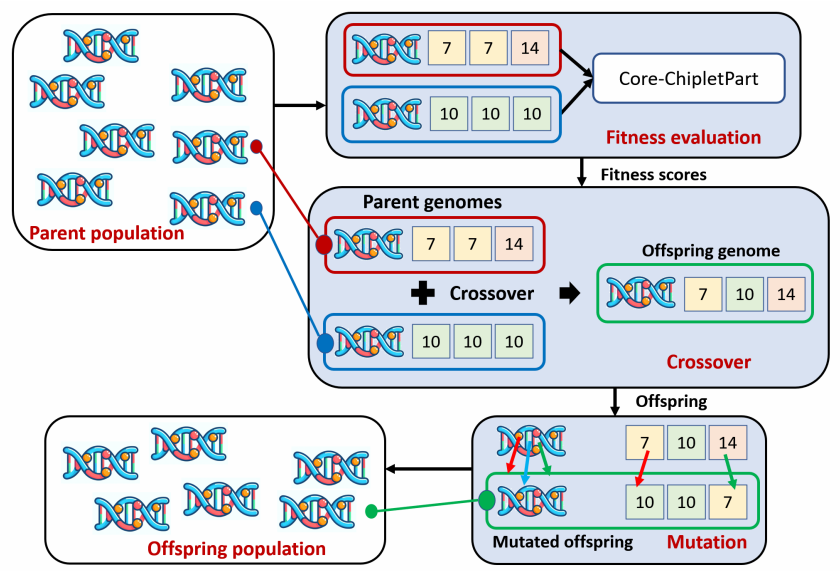

Fig. 3. GA-based technology assignment. A genome is a set of technology nodes. Shown: two parent genomes $\langle 7, 7, 14\rangle$ and $\langle 10, 10, 10\rangle$ undergo crossover to produce the offspring $\langle 7, 10, 14\rangle$; the offspring undergoes mutation to produce $\langle 10, 10, 7\rangle$ for the next-generation population.

- The improvement between successive generations falls below a predefined threshold, $\Delta_{threshold}$ ($\Delta_{threshold} = 0.01$ by default), and remains below this threshold for more than ϵ consecutive generations ($\epsilon = 10$ by default). If this condition is met, the algorithm terminates and returns C_{best} and ω_{best}.

Step 4: [Line 22] We use *tournament* selection [23] to identify candidate parents for crossover. Rather than evaluating the entire population at once, the selection process operates on randomly chosen "tournaments", each consisting of ζ genomes competing based on their fitness scores. The genome with the highest fitness in each tournament is selected as a candidate parent. This procedure is repeated until k_{pop} pairs of parents are chosen.[7]

Step 5: [Lines 23-25] We generate the next-generation population (offspring) through crossover, elitism, and mutation. Notably, we use uniform crossover [24] and random resetting mutation [25]. Additionally, we retain the top σ genomes from the parent generation based on their fitness scores (elitism), ensuring that high-quality solutions persist across generations.

In our implementation, we set $tot_{pop} = 50$, $\zeta = 3$, $k_{pop} = 45$, $\sigma = 5$ and $K_{max} = 8$ (Section 6.2.5). So, we conduct 90 tournaments (each tournament picks three genomes from the current generation) to select 90 parent genomes, which are then paired to generate 45 offspring genomes through crossover and mutation. We then supplement these 45 offspring with the top five genomes from the current generation (elitism) to generate the next generation population of 50 genomes. For additional implementation details, we refer the reader to [11].

Choice of optimizers: Our choice of picking GA and SA is motivated by their ability to flexibly optimize over *black-box*, non-convex and highly discontinuous cost landscapes — characteristics inherent to our chiplet cost model (Section 4.1). In contrast, ILP-based methods are less suitable for our framework due to the lack of closed-form or linearizable expressions for I/O-feasibility and chiplet

[7] A genome can be selected multiple times as a parent.

Algorithm 1: Overall *ChipletPart* framework.

Input: Standard Inputs: $\mathcal{T}$, $\mathcal{S}$
Hyperparameters: tot_{pop}, k_{pop}, $\Delta_{threshold}$, ϵ, ζ, Ψ, σ
Output: Chiplet partitioning solution C_{best},
Technology assignment ω_{best}

$\Delta_{threshold} \leftarrow 0.01$; $\epsilon \leftarrow 10$; $\zeta \leftarrow 3$; $\Psi \leftarrow 50$; $\sigma \leftarrow 5$;
$tot_{pop} \leftarrow 50$; $k_{pop} \leftarrow 45$; $p_c \leftarrow 0.60$; $p_m \leftarrow 0.07$; $K_{\max} \leftarrow 8$;
/* 1. Initial population generation */
$Cost_{best} \leftarrow \infty$; $Cost_{prev} \leftarrow \infty$; $\Delta \leftarrow \infty$;
Generate the initial population with tot_{pop} genomes where each genome is a random mapping from $K_{\max}$ chiplets to $\mathcal{T}$;
$i \leftarrow 0$; $num\_iters \leftarrow 0$;
while *true* **do**
  /* 2. Fitness evaluation */
  foreach ω_j *in the current population* **do**
    $C_{\omega_j} \leftarrow$ Generate the partitioning solution using *Core-ChipletPart* with technology assignment ω_j;
    $Cost_{\omega_j} \leftarrow$ Calculate the cost of C_{ω_j} (Section 4);
    if $Cost_{\omega_j} \leq Cost_{best}$ **then**
      $Cost_{best} \leftarrow Cost_{\omega_j}$; $C_{best} \leftarrow C_{\omega_j}$; $\omega_{best} \leftarrow \omega_j$;
  /* 3. Convergence check */
  if $num\_iters \geq \Psi$ **then**
    return C_{best}, ω_{best}
  $\Delta \leftarrow Cost_{prev} - Cost_{best}$;
  if $\Delta \leq \Delta_{threshold}$ **then**
    $i \leftarrow i + 1$;
    if $i > \epsilon$ **then**
      return C_{best}, ω_{best}
  else
    $i \leftarrow 0$;
  $Cost_{prev} \leftarrow Cost_{best}$;
  /* 4. Tournament selection */
  Select k_{pop} pairs of parents using tournament selection with tournament size ζ;
  /* 5. Crossover, elitism and mutation */
  Perform crossover on each selected pair (ω_x^p, ω_y^p) to generate offspring ω_x^o, using a crossover probability p_c;
  Apply mutation to each offspring ω_x^o with mutation probability p_m;
  Construct the next generation by selecting the top σ elite genomes from the current generation, along with k_{pop} offspring;

cost functions. GA enables direct co-optimization of technology assignment and chiplet partitioning, while SA ensures that floorplanning solutions are always I/O-feasible. Having a unified optimization approach and formulation is challenging since the solution space spans both discrete combinatorial variables (chiplet-to-tech assignment) and continuous geometric constraints (I/O reach), which are difficult to encode and optimize jointly. We hence opt for a modular decomposition — using GA for assignment and SA for feasibility evaluation. Note that Bayesian optimization (BO) is another powerful tool for optimizing complicated, black-box functions — we explore its potential as an alternative optimizer in Section 6.4.

In the following sections, we discuss the chiplet cost model, *Core-ChipletPart*, and the SA-based floorplanner.

4 CHIPLET PARTITIONING DRIVERS

Smaller chiplets can potentially bring lower costs and improved yield, while technology heterogeneity offers better power and performance. The overhead of inter-chiplet I/O also affects how a system should be partitioned into chiplets. In this section, we briefly discuss these factors.

4.1 Chiplet Cost Model

We use the open-source cost model from [9, 10]. It computes cost and yield based on the chiplet/interposer area (dependent on the technology node), assembly processes (dependent on bonding parameters), and I/O placement constraints (dictated by netlist connectivity and reach). The model calculates the individual cost and yield of each chiplet along with the interposer, and then aggregates them for the full assembly cost. We consider designs to be high volume in our studies, to minimize the impact of NRE. The cost model is summarized by Equation 1. For further details, see [9]. Benefits of using the cost model in [9] are discussed in Section 6.2 with results in Tables 5 and 6.

A block will have different areas and power in different technology nodes. While assigning a block to a more advanced technology node could, in principle, improve performance, we assume a fixed system architecture in this work and therefore keep performance constant while scaling power accordingly.[8] In this work, we follow [26] to model this scaling. Different technology nodes result in different cost per unit area for chiplets, along with differences in yield and reticle-fit dependent lithography costs [9]. We also scale memory and logic with different factors [27] since memory tends to scale poorly for advanced nodes. Different technology nodes will also have different non-recurring design [28] and manufacturing costs [29]. For example, the transition between 45 nm and 10 nm reduces logic size by a factor of ∼10× [26], while NRE design cost increases by ∼4.6× [28] and general cost per wafer increases by ∼2.6× [30].

4.2 Chiplet Power Model

We assume that performance of chiplets is preserved across technology nodes but that the power changes. We further assume that inter-chiplet communication latency is small enough to not influence the architecture. If certain inter-block interfaces are highly latency-sensitive, then those blocks should be merged in our framework to map them to the same chiplet. More complex performance models as in [7] are possible but not explored in this work.

Power is calculated as the sum of block power and I/O power. If all blocks are in the same chiplet, there will be no additional power due to I/O. However, if the blocks are placed into multiple partitions (chiplets), some additional power will be consumed by the added I/O cells. We scale the block power according to the scaling factors in [26] for different technology nodes.

4.3 I/O Reach Model

We use an I/O cell model where each I/O cell has both an area and a "reach" which is the maximum wire length that can be driven by the I/O cell. The reach depends on the I/O transceiver design and is often dictated by the inter-chiplet I/O standard. For example, *Universal Chiplet Interconnect Express* (UCIe) [31] guarantees a reach of 2 mm for advanced packages while standard packages (such as organic substrates) support larger reaches of up to 25 mm.

Since we assume a fixed system architecture, our partitioner does not modify the network structure or insert pipeline stages to compensate for long inter-chiplet wires. We also assume a standardized I/O cell with a fixed maximum reach instead of more complicated I/O structures. These assumptions help keep the partitioning problem well-scoped and tractable. We leave sophisticated I/O planning

[8] Modeling architectural changes based on block-to-technology assignments is beyond the scope of the current work and is left as a future direction.

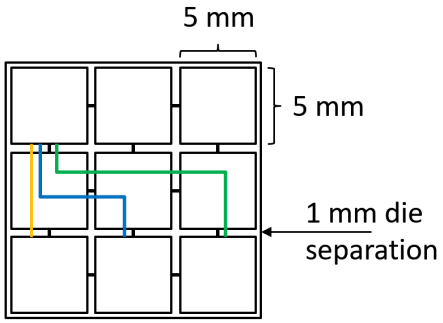

Fig. 4. Illustration of *reach*. Wirelengths: black - 1 mm, orange - 7 mm, blue - 13 mm, and green - 19 mm.

including pass-through connections, inserting buffers on the substrate, multiple I/O types per design, etc. to future work. However, users can explore multiple architectural variants and I/O types for the same design. For example, in our industry testcase (Section 6.1), we evaluate two versions that differ only in their crossbar configurations — and in Figure 14, we examine the impact of different I/O cell types on partitioning.

Consider the example in Figure 4, and assume that a bundle of wires is connected from the side of one chiplet to the side of another. If the chiplets are 5 mm on a side and spaced apart by 1 mm, then the short black connections require a reach of at least 1 mm, while the orange, blue, and green connections require reaches of at least 7 mm, 13 mm, and 19 mm respectively. Small values of reach constrain the floorplan, while larger values of reach come at the cost of more expensive I/O cells.

I/Os are discretized depending on the standard. For instance, a single UCIe I/O cell is as large as 0.88 mm^2 [31] while for parallel signaling (e.g., Advanced Interface Bus (AIB) or [32]), an I/O cell can be as small as 157 μm^2 [33]. This discretization can add complexity for chiplet partitioning, as the I/O (i.e., net cut) cost calculation follows a stepwise curve as a function of net cut.

5 CHIPLET PARTITIONING CORE

In this section, we first discuss our chiplet-cost-driven partitioning approach *Core-ChipletPart* in Section 5.1. Then, we present our reach-aware chiplet floorplanning approach in Section 5.2.

5.1 Cost-driven Partitioning: *Core-ChipletPart*

Our *Core-ChipletPart* does not adopt the widely-used multilevel approach since there are typically only a few hundreds of IP blocks in a block-level netlist. As shown in Figure 5, we (i) compute a large pool of initial partitioning solutions using a variety of graph partitioning algorithms; (ii) prune poor-quality partitioning solutions from the pool; and (iii) perform floorplan-aware Fiduccia–Mattheyses [34] (FM)-based and Kernighan-Lin [35] (KL)-based refinement. We then output the solution with the best cost.

Generation of a partitioning pool. Similar to the *initial partitioning* stage in widely-used multilevel partitioning approaches [4, 6], we compute a pool of initial partitioning solutions so as to explore a larger solution space.[9] Specifically, we use the following graph partitioning methods.

[9]However, unlike multilevel approaches [4, 6], *Core-ChipletPart* omits the coarsening stage due to the relatively small number of vertices (i.e., 384 IP blocks in our largest testcase, as shown in Table 4) in the netlist graph.

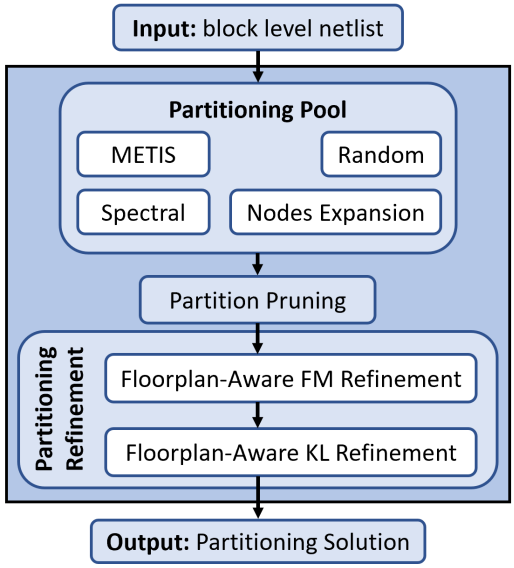

Fig. 5. *Core-ChipletPart* partitioning flow. We use multiple techniques to generate the initial partitions, then we prune out the worst-performing initializations before running refinement.

- *Spectral partitioning:* Because the block-level netlist is both small and sparse, we apply *spectral partitioning* [36]. We first compute the spectral embedding using the two smallest nontrivial eigenvectors, then cluster the IP blocks with the K-means algorithm [37].[10]
- *High-degree nodes expansion:* Our node expansion approach follows [39]. In the block-level netlist, high-degree nodes (e.g., crossbars) are distributed across different chiplets, and then expanded via breadth-first search. If multiple chiplets qualify for expansion, the block is assigned to the chiplet with which it has the strongest connection.
- *Random partitioning:* We randomly distribute all blocks across different chiplets. We generate multiple random solutions while sweeping $K = \{1, .., K_{\max}\}$.[11] We consider $K = 1$ to allow the degenerate case where all the blocks are placed in a single chiplet.
- *METIS:* We use the METIS graph partitioner [16] to create initial partitioning solutions, using the METIS APIs from [40]. Similar to our random partitioning approach, we sweep $K = \{2, .., K_{max}\}$.

Our partitioning pool consists of 11 solutions in total: one from the spectral method, one from node expansion, five from random partitioning, and four from METIS. We next discuss the pruning step, which discards low-quality solutions from this pool.

Partitioning solution pruning. We use a statistical filtering mechanism to prune the partitioning pool. First, we compute the cost of each initial partition using our cost model (Section 4.1), and then calculate the mean and standard deviation of these costs, as well as each partition's cost relative to the best solution. Our pruning applies two complementary thresholds: (i) a Z-score [41] threshold, eliminating partitions whose costs exceed 1.5 standard deviations above the mean and (ii) a relative

[10]In our implementation, we use a parallel K-means method with smart initialization (K-means++ [38]), and we set $K = 4$ by default.

[11]Our studies show that $K_{\max} = 8$ is sufficient for our testcases. If the solution finds $K_{\max}$ number of partitions, the user can increase $K_{\max}$ to check larger numbers of partitions.

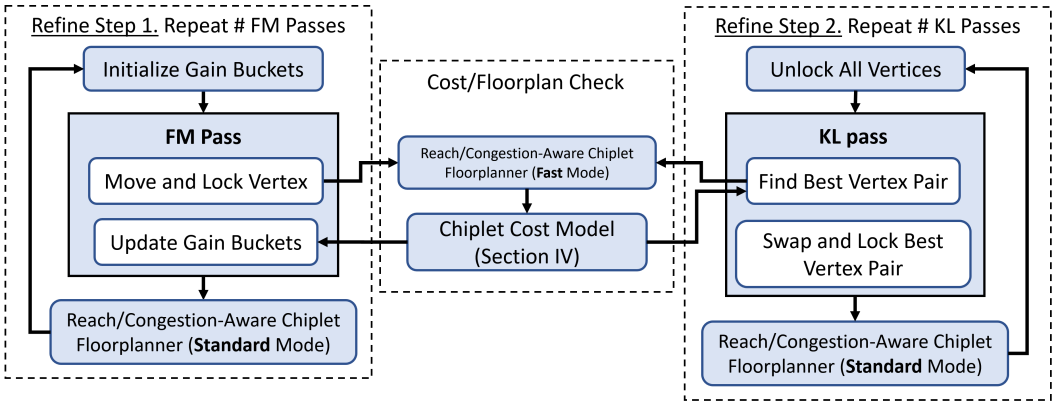

Fig. 6. Floorplan-aware FM and KL-based refinement. In our implementation, we run FM first, followed by KL.

cost threshold, removing partitions whose costs are worse than twice the minimum cost. To prevent over-pruning, we retain at least three solutions even if they all appear statistically poor. Pruning significantly reduces computational workload by eliminating low-quality candidates early.

Partitioning refinement. We use two strategies for refinement: (i) FM and (ii) KL. Both of these processes are shown in Figure 6. Each potential FM move or KL swap triggers a call to the chiplet cost model to obtain the cost (gain) of an updated partitioning solution. This evaluation includes generating a floorplan solution using the "fast" mode of our SA-based reach-aware chiplet floorplanner, followed by calculating the corresponding cost. After completing each FM or KL pass, the "standard" mode of the SA-based reach-aware chiplet floorplanner is employed to search for an improved floorplan solution for the subsequent pass. Further details about our SA-based reach-aware chiplet floorplanner are provided in Section 5.2. We adopt the K-way FM implementation from *TritonPart* [6], while our KL refinement [42] follows the methodology described in [35]. In our implementation, we apply KL after FM to further improve solution quality, as KL explores a broader neighborhood by enabling pairwise vertex swaps — compared to FM's single-vertex moves.

5.2 Reach-aware Chiplet Floorplanning

Chiplet floorplanning is a crucial step, as it determines the location and shape of each chiplet on the interposer. In contrast to the well-studied macro placement, chiplet floorplanning must also consider *I/O-feasibility*. More specifically, the wirelength of a net must not exceed the reach specified by the corresponding I/O cell (Section 4). Figure 7 illustrates the wirelength (measured in terms of Manhattan distance) calculation for net e connecting chiplets A and B. Assuming that the *bitwidth* of net e is n_e and the area of I/O cells is A_{IO}, then the wirelength of net e ($length(e)$) in the figure is calculated as: (i) $l = \sqrt{w^2 + 2A_{IO}} - w$ and (ii) $length(e) = h + 2l$, where l is the *depth* needed for all the I/O cells.[12] Then the reach violation penalty for net e is $n_e \times max(length(e) - reach(e), 0.0)$. The reach violation penalty for a chiplet is defined as the summation of the reach violation penalties for all the nets connected to the chiplet.

Problem formulation. Our work uses a novel reach-aware chiplet floorplanning approach. The input to our floorplanner is a set of chiplets C, and a chiplet-level netlist $\mathcal{N}$ that defines the connections between chiplets. The output is a floorplanning solution which provides the location and shape (width

[12]The detailed code for more general cases is available in [43].

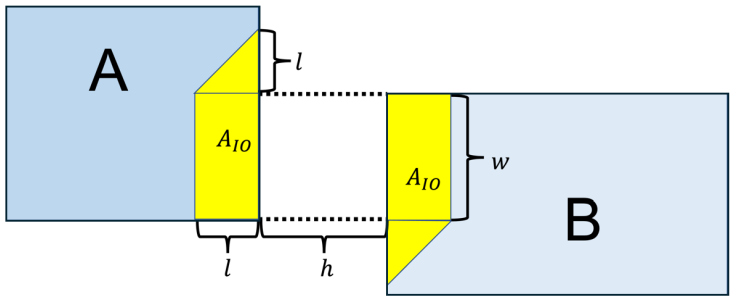

Fig. 7. Illustration of wirelength computation ($length(e)$). The yellow region indicates the placement area for I/O cells that will satisfy the reach constraint.

and height) of each chiplet. The objective of our floorplanner is

$$WL_{reach} = \sum_{e \in N} n_e \times max(length(e) - reach(e), 0.0) \tag{2}$$

$$\textbf{min} \quad \alpha \times WL_{reach} + \beta \times A_C + \gamma \times A_P \tag{3}$$

where WL_{reach}, A_C, and A_P respectively denote the reach violation penalty, the area of chiplets, and the area of the package. α, β and γ are user-defined coefficients that can be modified to achieve a desired trade-off between the different objectives in Equation 3.[13] Note that we allow the area of chiplets to increase to satisfy the reach constraints. During optimization, the following constraints are considered:

- **Overlap constraint**: No two chiplets can overlap.
- **Separation constraint**: In practical chiplet-based systems, extra space must exist between neighboring chiplets to account for dicing and alignment accuracy, and to prevent defects and mechanical stress during chiplet assembly. The separation constraint sets the minimum distance between any two chiplets.

SA-based chiplet floorplanner. Our reach-aware floorplanner uses Sequence Pairss [44] to represent a spatial arrangement of chiplets in the netlist and Simulated Annealing [45] to optimize the objective function (Equation 3). Our annealer supports five solution perturbation (move) operators, each selected with equal probability (0.2).

- **Op1:** Swap two chiplets in the first sequence.
- **Op2:** Swap two chiplets in the second sequence.
- **Op3:** Swap two chiplets in both sequences.
- **Op4:** Reshape a chiplet. Identify the chiplet with the highest reach-violation penalty and modify its shape using the resizing algorithm described in [46]. Figure 8 (top) illustrates how a reach violation for the net connecting chiplets E and B (red dashed arrow) is resolved by aligning the right boundary of B with that of E.
- **Op5:** Expand (i.e., bloat) a chiplet. Identify the chiplet with the highest reach-violation penalty and expand it into neighboring whitespace, where the expansion in each direction is determined by the extent required to eliminate the violation. Figure 8 (bottom) shows how reach violations for nets connecting chiplets B–G and B–F are resolved by expanding B toward the top and right. This operator reduces reach-violation penalties at the cost of increasing the chiplet area and potentially the overall package area as well.

[13]The default values for α, β and γ are 1.0, 1.0 and 1.0, respectively.

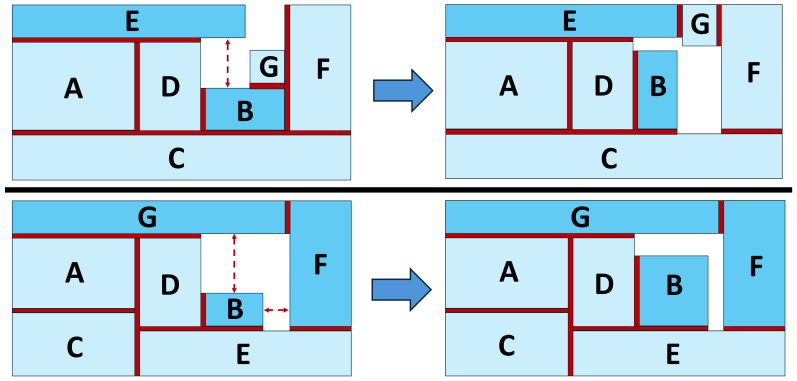

Fig. 8. Top: Resizing of chiplet B fixes a reach violation for a net connecting chiplets B-E. Bottom: Expansion (bloating) of chiplet B fixes reach violations for nets connecting chiplets B-G and B-F. In each figure pair, left = before; right = after. Red regions indicate the separation constraint between chiplets, and red dashed arrows indicate reach violations before fixing.

To enhance the performance of simulated annealing (SA), we adopt a *Go-With-the-Winners* (GWTW) strategy [22], which enables 10 parallel SA *walkers* to independently explore the solution space. Periodically, a synchronization phase is performed in which: (i) the best-performing threads are identified, (ii) their solutions are cloned to repopulate the thread pool and (iii) all threads resume independent exploration. This strategy balances diversification and intensification, improving convergence without incurring significant runtime overhead. We use 10 threads in our experiments, a setting that is easily supported on standard server-class machines.[14] As discussed in Section 5.1, our chiplet floorplanner operates in two modes: "standard" and "fast". The standard mode performs 1,000,000 perturbations with a cooling rate of 0.989, while the fast mode performs 10,000 perturbations with an initial temperature of 0.0 (greedy). In our experiments, the "standard mode" is invoked after each FM or KL pass (default: 4 passes), totaling four invocations per run. The fast mode is triggered after each FM vertex move (default: 50% of all vertices) and each KL vertex swap (default: 10% of all vertices), resulting in 230 invocations per pass in our largest testcase. These parameters are empirically selected to maintain a balance between runtime and solution quality.

6 EXPERIMENTAL RESULTS

ChipletPart is implemented using approximately 27K lines of C++ code, building upon implementations from [6]. We use OpenMP [47] for parallelization, the Eigen library [48] for spectral partitioning, codes from [40] to run METIS, and Boost [49] for high precision arithmetic computations in the cost model. Additionally, we have translated the Python-based cost model from [9, 10] into C++ and integrated it into our *ChipletPart* framework. The scripts and source code of *ChipletPart*, including its cost model implementation, are available in [11]. We run all experiments on a Linux server with Intel Xeon E5-2690 CPU (48 threads) and 256GB RAM.

6.1 Benchmarks and Baselines

We evaluate four categories of testcases in this work: *Waferscale*, *MemPool*, *Industrial Testcase*, and *Comparison Testcases*. A common issue in chiplet partitioning research is that most testcases are too small to warrant practical chiplet partitioning; existing works often focus on very small designs,

[14]This number of threads is typical for modern multi-core servers and does not impose a significant hardware burden.

overemphasizing the impact of I/O and underemphasizing yield and packaging costs. To address this, we scale up the power and area of our baseline designs to sizes more representative of real commercial systems (e.g., Intel's Ponte Vecchio [1]). We also scale the interconnects using Rent's Rule,[15] applying constants from [51] for microprocessor and memory blocks. This ensures that our scaled designs have a realistic number of I/Os, making them suitable for evaluating chiplet partitioning.

Our first category of testcases (*Waferscale*) is derived from the waferscale graph processor in [32]. We extract synthesized IP block areas and a block-level netlist at 45 nm, then apply area and power scaling to favor partitioning. The second category (*MemPool*) is similarly extracted from synthesized blocks in [52] and likewise scaled from its 45nm block-level netlist. The third category (*Industry Testcase*) is based on a 16nm design provided by a semiconductor company, which we also scale to a suitable size. Finally, the *Comparison Testcases* are based on publicly available information for large commercial designs, and come from [8]. The original implementations span multiple technology nodes, and these designs are sufficiently large that no additional scaling is required.

To ensure that routing congestion is not a concern in our example systems, we performed a simple analytical check based on estimated routing capacity and required wires count. We considered both *escape routing* (breakout of signals from each chiplet on the substrate) and *global routing* (inter-chiplet connections across the substrate). In all experiments, we considered a substrate with four routing layers and a routing pitch of 1μm. This is similar to existing technologies [53]. For escape routing, we verified whether the number of available routing tracks around each chiplet's perimeter is sufficient to accommodate all connections to that die. For global routing, we examined the vertical slices across the substrate and confirmed that each slice contained enough routing tracks to support all inter-chiplet connections crossing that slice. Additionally, we used alternating routing directions across routing layers for global routing. Under these considerations, we did not observe routing congestion in any of our testcases.

Table 4 shows the specific testcase configurations that we use; see also the design-specific discussion below. In the table, the *WS* testcases are based on the *Waferscale* design, the *MP* testcase is based on the *MemPool* design, the *TC* testcase is based on the *Industry* design, and *EPYC* and *GA100* are the *Comparison Testcases*. The specific design configurations are explained as follows.

Waferscale Testcases. The *Waferscale* design is organized into a grid of tiles connected in a 2D mesh configuration with neighbor-to-neighbor communication between tiles. Each tile comprises 48 IP blocks. Those blocks include a router for tile-to-tile connections, a crossbar for intra-tile connections, four large shared memory blocks connected to the crossbar, and 14 cores each with associated bus and private memory blocks. Figure 9 gives a system block diagram. We have implemented a testcase generator (see [11]) which allows us to choose the number of tiles in a given testcase, along with the number of cores and memories per tile. Area and power scaling are also provided via the testcase generator. As noted, Table 4 shows the *Waferscale* configurations used in our case studies.

MemPool Testcase. The full *MemPool* design consists of four *MemPool groups*, each containing 16 *MemPool tiles*. These tiles each consist of cores and memory along with supporting logic. Note that while each tile contains four cores and 16 memory banks, we do not split tiles for partitioning. For our *MemPool* testcase, we use a single *MemPool* group, i.e., taken at just above the tile level of the hierarchy. Based on 16 tiles and 24 blocks related to (remote, local, AXI etc.) interconnect, our block-level netlist for *MemPool* group has 40 IP blocks to use in partitioning. A diagram is available in [11]; see also Figure 3(a) in [52].

[15]Rent's Rule [50] states that the number of terminals T in a logic block scales as $T = kC^p$, where k is a constant, C is the number of components and p is the Rent parameter. Hence, if C is scaled by a factor s, T is scaled by s^p.

Table 4. Benchmark characteristics. Note that MP and WS both use tile terminology; the others do not.

Benchmark	# Tiles	# IP Blocks	Area (mm^2)	Area Scaling	Power Scaling
WS_1	1	48	1582 (45nm)	1600	1600
WS_2	2	96	3165 (45nm)	1600	1600
WS_3	4	192	6330 (45nm)	1600	1600
WS_4	8	384	12,660 (45nm)	1600	1600
MP	16	40	494 (45nm)	100	100
TC_1	N/A	14	567 (16nm)	16	16
TC_2	N/A	17	567 (16nm)	16	16
EPYC	N/A	32	148 (7nm)	1	1
GA100	N/A	180	548 (7nm)	1	1

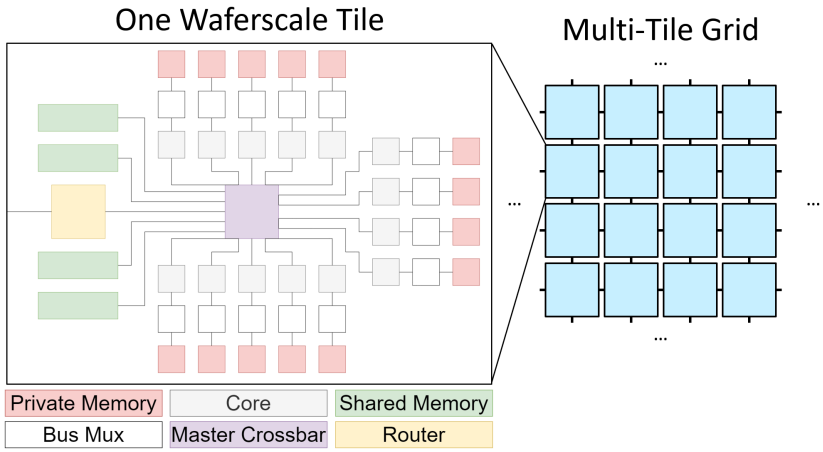

Fig. 9. *Waferscale* testcase. Left: example IP blocks in a tile. Right: Connection of tiles in a grid. This testcase allows scaling of the number of tiles to match a target design size for partitioning.

Industry Testcases. We use a chip design from industry with parameters shown in Table 4.[16] Because the original design is relatively small, we scale the area and interconnect via Rent's Rule to reach an overall design area of more than 400 mm^2. This architecture follows a controller-target structure around a crossbar, where the crossbar constitutes 30% of the total area of its connected blocks.

Comparison Testcases. We also evaluate on two commercial designs used by *Chipletizer* [8]. The first is based on AMD's *EPYC* 7282 [55], shown in Figure 10. We additionally examine NVIDIA's *GA100* chip [56], which powers the A100 GPU. The *GA100* design includes 128 simultaneous multiprocessor blocks, 40 L2 blocks and multiple HBM controller blocks, totaling 180 blocks. We omit the *GA100* block diagram due to its size.

Metrics. For simplicity, we assume iso-performance for our testcases, even across technology nodes; instead, we derive advantages in power from improvements in I/O count or changes in technology node. We also assume that connections between blocks in a chiplet will have the same speed as

[16]The industry testcases are generated based on discussions with Analog Devices engineers [54].

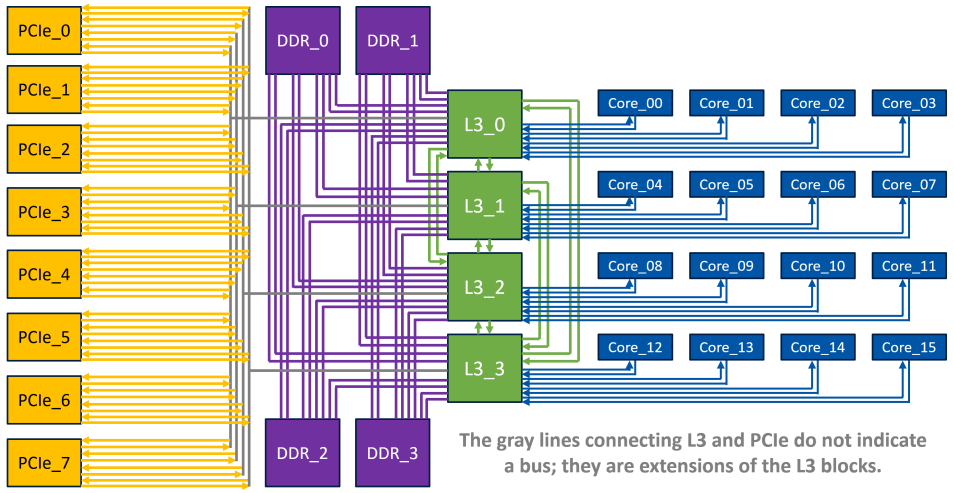

Fig. 10. AMD EPYC 7282 Testcase used in [8].

inter-chiplet connections, although the inter-chiplet connections will have higher power and area requirements due to large I/O drivers. Design area is a major contributor to cost, and chip power contributes to chip cost via added bumps for current delivery that require additional area. For this reason, we use cost as a single metric for partitioning.

Baselines. Since there is no open-source, cost-driven chiplet partitioner available, we compare against: (i) *Min-cut partitioners* (*hMETIS* [16], which is often used in chiplet flows [15], and *TritonPart* [6]) applied to weighted hypergraphs, where each hyperedge is weighted by the corresponding I/O bandwidth; (ii) the *parChiplet* method from *Floorplet* [7]; (iii) *Chipletizer* [8], for which we thank the authors for providing access to their code; and (iv) partitioning solutions generated by human engineers.

6.2 Validations of Chiplet Partitioning

Given that previous works use traditional min-cut partitioners in 2.5D flows, we directly compare *ChipletPart* with leading partitioners *hMETIS* and *TritonPart*. We also compare to the *Floorplet* floorplan-aware chiplet partitioner [7], *Chipletizer* [8], and *manual* partitions. For the *Waferscale* testcases based on [32] we compare the memory-compute partition in that work with a per-tile partitioning as the manual partition. Overall, our validations span (i) validation of chiplet cost-driven partitioning, (ii) validation of floorplan-awareness, (iii) validation of heterogeneous technology-awareness, and (iv) analyses of hyperparameter sensitivities and runtime.

6.2.1 Validation of chiplet cost-driven partitioning. We evaluate *ChipletPart* against the baselines listed in the previous section: *hMETIS*, *TritonPart*, *Floorplet*, *Chipletizer*, and manually derived partitions. Our validation is thus divided into three parts, detailed below.

[17]For the six testcases where *Chipletizer* produces solutions, the geometric mean cost (GM cost) is 48.5 vs. 33.9 for *ChipletPart*, with *ChipletPart* achieving a geo-mean cost improvement (GM imprv.) of 30.1%.

Table 5. Partitioning costs and number of chiplets for hMETIS, TritonPart, Floorplet [7], Manual, Chipletizer [8], ChipletPart$_{\text{No-FP}}$, and ChipletPart. Best cost per benchmark is in bold. We additionally report the geometric mean (GM) of absolute costs as well as the GM percentage improvement relative to the Manual baseline. Positive GM improvement indicates lower (better) cost on average. Chipletizer results are shown only for testcases where valid solutions were obtained. Because these cover a subset of the full design suite, GM values computed over six testcases are not directly comparable to the nine-testcase GM; hence, we denote Chipletizer's GM cost (and improvement %) as N/A.[17]

Benchmark	hMETIS		TritonPart		Floorplet		Manual		Chipletizer		ChipletPart$_{\text{No-FP}}$		ChipletPart	
	$\|C\|$	Cost	$\|C\|$	Cost	$\|C\|$	Cost	$\|C\|$	Cost	$\|C\|$	Cost	$\|C\|$	Cost	$\|C\|$	Cost
WS_1	5	72.4	3	77.3	6	55.5	1	71.2	1	71.2	7	55.1	6	**53.9**
WS_2	8	138.1	7	143.2	4	150.3	2	145.5	Fail	Fail	6	122.3	8	**123.4**
WS_3	7	316.3	8	362.4	4	310.5	4	310.5	Fail	Fail	8	319.6	7	**300.4**
WS_4	8	1449.5	8	1564.2	5	1237.9	8	674.3	Fail	Fail	7	776.2	8	**659.9**
MP	1	5.7	1	5.7	1	5.7	1	5.7	1	5.7	1	5.7	1	**5.7**
TC_1	3	72.6	3	68.1	6	45.6	1	70.5	1	70.5	4	47.2	6	**46.5**
TC_2	3	73.9	3	71.1	7	46.5	4	49.4	1	74.9	6	48.7	7	**46.2**
EPYC	4	92.1	4	96.3	4	89.3	4	80.8	6	139.3	8	76.1	8	**72.5**
GA100	3	52.3	3	60.5	2	42.1	5	36.1	5	43.8	2	33.6	5	**31.8**
GM cost	–	**148.3**	–	**171.0**	–	**113.7**	–	**114.9**	–	**N/A**[17]	–	**85.6**	–	**82.3**
GM impr. (%)	–	**-21%**	–	**-26%**	–	**-2%**	–	**0%**	–	**N/A**[17]	–	**+9%**	–	**+13%**

Table 6. Chiplet count and I/O-feasibility for hMETIS, TritonPart, Floorplet [7], Manual, Chipletizer [8], ChipletPart$_{\text{No-FP}}$, and ChipletPart. "Feas" indicates I/O-feasibility (✓: feasible, X: not feasible; "Fail" indicates that no valid solution was obtained). We also report the number of successful runs for each category.

Benchmark	hMETIS		TritonPart		Floorplet		Manual		Chipletizer		ChipletPart$_{\text{No-FP}}$		ChipletPart	
	$\|C\|$	Feas	$\|C\|$	Feas	$\|C\|$	Feas	$\|C\|$	Feas	$\|C\|$	Feas	$\|C\|$	Feas	$\|C\|$	Feas
WS_1	5	X	3	✓	6	X	1	✓	1	✓	7	X	6	✓
WS_2	8	X	7	X	4	✓	2	✓	Fail	Fail	6	X	8	✓
WS_3	7	X	8	X	4	✓	4	✓	Fail	Fail	8	X	7	✓
WS_4	8	X	8	X	5	X	8	✓	Fail	Fail	7	X	8	✓
MP	1	✓	1	✓	1	✓	1	✓	1	✓	1	✓	1	✓
TC_1	3	✓	3	✓	6	X	1	✓	1	✓	4	✓	6	✓
TC_2	3	✓	3	✓	7	X	4	✓	1	✓	6	X	7	✓
EPYC	4	✓	4	✓	4	✓	4	✓	6	X	8	X	8	✓
GA100	3	✓	3	✓	2	✓	5	✓	5	X	2	✓	5	✓
#Successes (out of 9)	4		6		5		9		4		3		**9**	

Comparison with netlist partitioners. We evaluate the chiplet cost-driven partitioning capabilities of *ChipletPart* by benchmarking against leading netlist partitioners *hMETIS* and *TritonPart*, using the latter's default parameter settings.[18] We use the testcases listed in Table 4 for our evaluations, with results presented in Tables 5 and 6. Table 5 presents chiplet cost comparisons and Table 6 presents floorplan-feasibility comparisons. For both *hMETIS* and *TritonPart*:

- We set an imbalance factor of 5% and define our input set of partitions as $K = 1, 2, 3, ..., 10$.
- For each K, we execute ten runs of the partitioner, yielding a total of 100 partitioning solutions.[19]

[18]For *hMETIS*, the default settings are Nruns = 10, CType = 1, RType = 1, Vcycle = 1, Reconst = 0, and seed = 0 [4]. For *TritonPart*, the default parameter settings are $thr\_coarsen\_hyperedge\_size\_skip = 200$, $coarsening\_ratio = 1.6$, $max\_moves = 60$, and $num\_coarsen\_solutions = 3$ [6].

[19]Our studies show that longer partitioner runs do not improve the quality of results.

- We evaluate each partitioning solution using our cost model (Secs. 4 and 4.2) and report the solution with lowest chiplet cost.
- Since *hMETIS* and *TritonPart* are not technology-aware, for a fair comparison, we enforce homogeneity by running *ChipletPart* with only the 7 nm technology node. Consequently, the cost model evaluates all partitioning solutions using the 7 nm technology node. The floorplanner is run in the standard mode (Section 5.2).

Results are presented in Tables 5 and 6. Columns $|C|$, $Cost$, and $Feas$, respectively indicate the number of chiplets, the evaluated cost of the chiplet solution, and whether the solution satisfies the reach constraints. *ChipletPart* consistently produces solutions with better cost than *hMETIS* and *TritonPart* with improvements of up to 55% and 58% respectively. The geometric mean improvements over *hMETIS* and *TritonPart* are 16% and 20%, respectively. This result shows the advantages of *ChipletPart*'s cost-driven partitioning over traditional min-cut partitioners. Furthermore, *ChipletPart* consistently produces floorplan-feasible solutions, whereas standard netlist partitioners often fail to meet the reach constraints.[20]

Comparison with previous work. We compare *ChipletPart* with the *parChiplet* partitioner from *Floorplet* [7], and *Chipletizer* [8]. Here, we set the *Floorplet* parameters as $|C|_{\min} = 1$, $|C|_{\max} = 10$ and $rr = 1.0$. All partitioning solutions are evaluated using our cost model with the 7 nm technology node and the floorplanner is run in standard mode. Tables 5 and 6 present these results. Compared to *Floorplet*, *ChipletPart* achieves up to a 47% cost improvement, with a geometric mean improvement of 6%. While *Floorplet* produces a solution with slightly lower cost on the TC_1 testcase, the solution violates reach constraints. In contrast, *ChipletPart* consistently generates floorplan-feasible solutions with better cost, highlighting its strength in delivering high-quality and physically realizable chiplet partitions. For *Chipletizer*, we use the default settings from the original implementation, with *production volume* [8] set to 500,000 units. *Chipletizer* fails to produce a solution on three of nine testcases (WS_1, WS_2, WS_3, WS_4) due to scalability limitations. Across the remaining testcases, *ChipletPart* achieves up to 48% lower cost[21] compared to *Chipletizer*, while consistently generating I/O-feasible solutions. For example, on *GA100*, both frameworks produce a five-chiplet solution, but *ChipletPart* achieves a 27% cost reduction and satisfies I/O-feasibility, whereas the *Chipletizer* solution violates feasibility.

Comparison with human baselines. We generate human baselines for each testcase as follows. For the *Waferscale* (WS_1, WS_2, WS_3, WS_4) designs, which are tile-based, the human partitioning solution aligns directly with these tiles. For the *red*4-way partition with the same blocks as TC_1 but with a 4-way split crossbar. For the *EPYC* testcase (Figure 10), we cluster each set of cores with its corresponding L3 — distributing other blocks to maintain regularity and balanced area — resulting in four partitions each containing two PCIe blocks, one DDR, one L3, and four cores. Similarly, for *GA100*, we create five chiplets: four partitions each holding 32 simultaneous multiprocessors and 10 L2 blocks, and one partition for the remaining blocks.[22]

Similar to the previous sections, all partitioning solutions are evaluated using our cost model with the 7nm technology node and the floorplanner is run in standard mode. Tables 5 and 6 present the results. *ChipletPart* generates solutions that are up to 34% better than the manual baselines with a

[20] *hMETIS* and *TritonPart* optimize cutsize and return partitions with better cutsize compared to *ChipletPart* (albeit with worse chiplet cost).

[21] A more effective way to run *Chipletizer* would be to integrate it with our cost model. Since this is beyond the scope of our current work, we leave it for future research.

[22] Our human baselines do not necessarily reflect partitions chosen by industry. Instead, we split the blocks to generate relatively even partitions with small cutset and high regularity as an example of what the human engineer might decide to do.

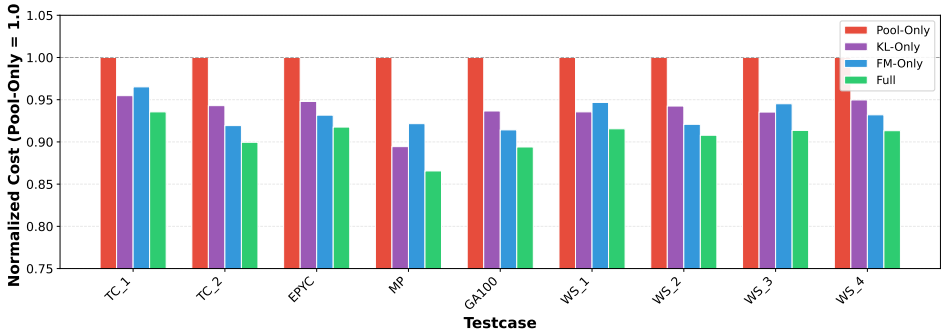

Fig. 11. Impact of initial partitioning and refinement in *Core-ChipletPart*. *Pool-Only* denotes *Core-ChipletPart* with no refinement. *KL-Only* denotes *Core-ChipletPart* with KL refinement only. *FM-Only* denotes *Core-ChipletPart* with FM refinement only.

geometric mean improvement of 13%. While the human baselines are all floorplan-feasible, they are not cost-optimal. These results indicate that a black-box partitioner such as *ChipletPart* can generate higher-quality solutions than manual partitioning strategies.

6.2.2 Validation of floorplan-awareness. We evaluate the impact of floorplan-awareness in *ChipletPart*. Tables 5 and 6 compare solutions generated with and without floorplan-awareness enabled, denoted as *ChipletPart* and $ChipletPart_{No\text{-}FP}$ respectively. Disabling floorplan-awareness can occasionally yield lower nominal chiplet cost (e.g., WS_2), but it frequently produces solutions that violate interconnect reach constraints. In particular, six out of nine testcases result in infeasible solutions when floorplan-awareness is disabled. Beyond feasibility, enabling floorplan-awareness improves chiplet cost by up to 15%, with a geometric mean improvement of 4%. This is because our cost model incorporates floorplan-level information when evaluating partition quality — thus, floorplan-aware solutions better align with the true cost objective. These observations highlight the importance of incorporating floorplan-awareness: while it may slightly increase nominal cost in some cases, it helps to ensure that all generated solutions are physically realizable.

6.2.3 Impact of initial partitioning and refinement. To quantify the contribution of each component in *Core-ChipletPart*, we conduct an ablation study comparing four configurations: (i) the *full* framework i.e., *Core-ChipletPart*; (ii) initial partitioning *pool-only* with no refinement; (iii) initial partitioning pool with FM refinement only; and (iv) initial partitioning pool with KL refinement only. Figure 11 presents the results across all testcases from Table 4. The full *Core-ChipletPart* achieves an average cost reduction of 9.3% compared to the *pool-only* baseline, with improvements ranging from 6.4% to 13.4% across different designs. When examining the individual refinement strategies, *FM-only* achieves 6.4% average improvement while *KL-only* achieves 5.7% average improvement over the pool-only baseline. Notably, neither refinement strategy consistently dominates the other: *FM-only* outperforms *KL-only* on five out of ten testcases, while *KL-only* performs better on the remaining four testcases. This complementary behavior demonstrates that the two refinement algorithms optimize different aspects of the partitioning solution. FM excels at move-based local optimization while KL provides effective swap-based refinement for certain design topologies. Importantly, the *full Core-ChipletPart* consistently outperforms both individual refinement strategies across all testcases, achieving an additional 2-3% improvement over the better of *FM-only* or *KL-only* in each case. We believe that these results validate our integrated approach: the diverse initial partition pool provides high-quality starting points, while the application of FM and KL refinement generates further improvement that neither algorithm achieves independently.

Table 7. Impact of heterogeneity. "Homogeneous" denotes homogeneous integration cost. "Heterogeneous" denotes heterogeneous integration cost, chiplet tech distribution, and *ChipletPart* runtime. A single invocation of *Core-ChipletPart's runtime is* ∼5% of *ChipletPart*'s runtime. Runtimes reported are the CPU time.

Bench.	Homogeneous			Heterogeneous				
	7nm	10nm	14nm	Cost	#7nm	#10nm	#14nm	Runtime (s)
WS_1	53.9	48.9	42.1	**41.7**	2	0	4	401
WS_2	123.4	112.7	109.4	**87.8**	1	0	7	1421
WS_3	300.4	310.4	286.3	**254.4**	3	0	5	3240
WS_4	659.9	662.4	676.1	**656.2**	7	1	0	5624
MP	5.7	6.9	8.8	**5.7**	1	0	0	180
TC_1	46.5	52.4	56.4	**39.8**	3	0	2	160
TC_2	46.2	54.2	64.1	**39.3**	4	0	2	89
EPYC	72.5	86.2	94.6	**65.1**	3	0	2	349
GA100	31.8	40.5	55.2	**31.6**	3	1	0	493

6.2.4 Validation of heterogeneous technology-awareness. We evaluate benefits of incorporating heterogeneous technology awareness on all nine testcases (Table 4) and a technology list comprising 7 nm, 10 nm, and 14 nm nodes. Table 7 reports chiplet cost reductions achieved by enabling technology awareness, relative to the best solutions obtained under homogeneous technology integration for each node. Specifically, we first run *ChipletPart* using a single technology node per testcase (homogeneous integration) to obtain baseline partitioning solutions for each technology. We then run *ChipletPart* with technology awareness enabled (i.e., allowing the use of all three technology nodes) and compare the resulting solutions. Our results show that heterogeneous technology assignment can reduce system cost by up to 43%, with geometric mean cost reductions of 7%, 15%, and 15% when compared against homogeneous solutions at 7 nm, 10 nm, and 14 nm, respectively.[23] Note, that all solutions generated by *ChipletPart* are floorplan-feasible.

6.2.5 Hyperparameter exploration and sensitivity analysis. *ChipletPart*'s genetic algorithm relies on four key hyperparameters: (i) total population size tot_{pop}; (ii) maximum number of generations Ψ; (iii) mutation probability p_m; and (iv) crossover probability p_c (cf. Algorithm 1).[24] We validate the default values of these hyperparameters via an empirical study using testcases WS_1, WS_2, TC_1, and TC_2 from Table 4. We use the following technology nodes for this study: 7 nm, 10 nm, and 14 nm, and run *ChipletPart* on the four aforementioned testcases. Figure 12 presents the normalized chiplet costs (normalized to the cost of *manual* solutions) and the normalized runtime (represented as "Normalized Value" in Figure 12), normalized to a constant value of 10000 seconds.[25] In our experiments, we systematically vary each hyperparameter while keeping the others fixed at their default values. For each configuration, we measure the resulting chiplet cost and runtime across the testcases. Our observations from Figure 12 indicates that the default hyperparameter settings achieve a balanced trade-off between partitioning quality and runtime efficiency.

[23] To further validate the effectiveness of our GA, we implemented an enumeration-based framework that exhaustively explores all $\sum_{k=1}^{K_{\max}} \binom{k+m-1}{m-1}$ possible assignments. Our experiments show that the GA achieves up to 7× speedup with less than 1% degradation in chiplet cost compared to the enumeration-based framework.

[24] *ChipletPart* never returns a chiplet solution with more than 8 chiplets, so we set K_{max} to 8.

[25] 10000 seconds is chosen for normalization to allow for comparison with about three hours of runtime.

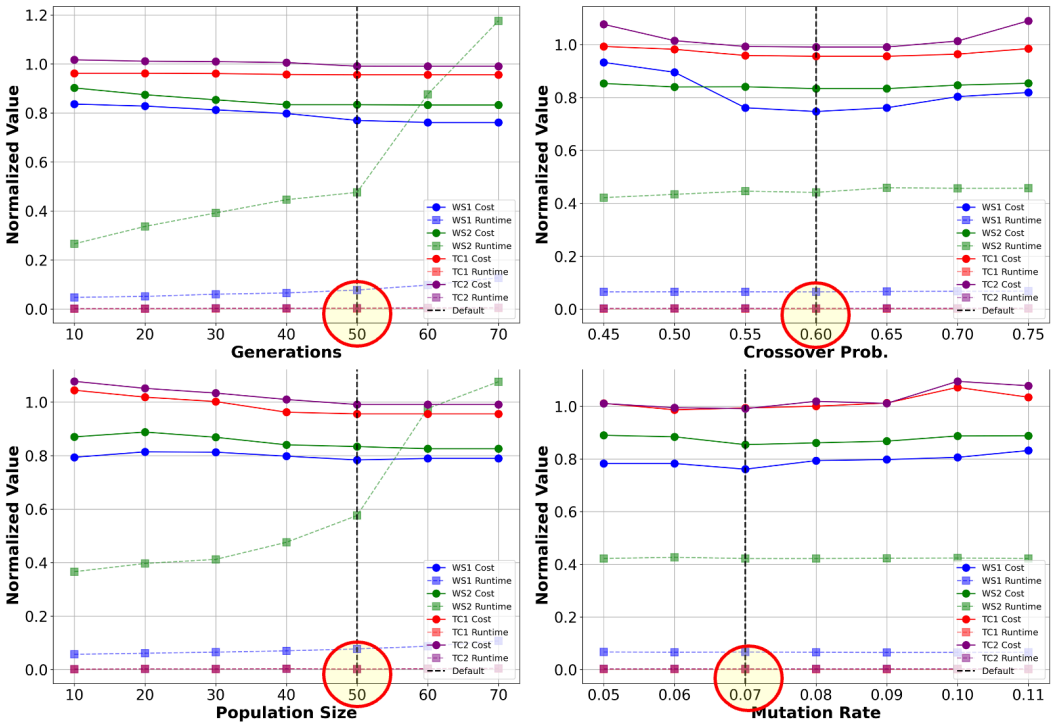

Fig. 12. Hyperparameters sweep across benchmarks. Solid lines denote normalized cost and dashed lines denote normalized runtime. The default hyperparameter choice is marked with a vertical dashed line and circled.

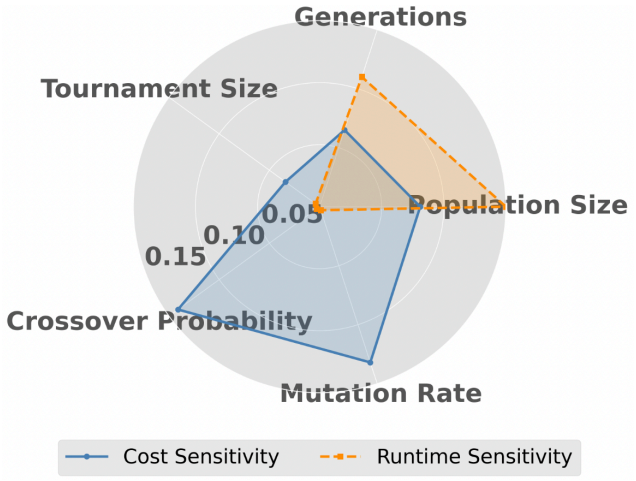

Fig. 13. Sensitivity analysis of hyperparameters. Larger distances from the plot center indicate greater influence.

Table 8. Runtime breakdown of *ChipletPart*.

Component	Runtime share (%)
SA-based floorplanner	67
Partition refinement (chiplet cost model)	28
Partition pool	2
Partition pruning	1
File I/O handling	1

Figure 13 summarizes the sensitivity of *ChipletPart* to its hyperparameters. We measure sensitivity as the normalized variance in cost and runtime across the sweep range for each hyperparameter. Notably, crossover and mutation probability have the highest influence on cost, while population size and number of generations dominate runtime variation. Tournament size has relatively low sensitivity for both metrics. These results indicate that careful tuning of crossover and mutation parameters is essential for chiplet solution quality, while population size control primarily affects runtime overhead.

6.3 Runtime remarks

ChipletPart is implemented entirely in C++, including a high-quality C++ translation of the Python-based cost model from [10]. Our optimized C++ translation achieves approximately a 5× speedup compared to the original Python version. Empirical results indicate that *ChipletPart*, when integrated with the translated C++ cost model, attains over a 100× speedup relative to an integration based on the Python cost model. Additionally, our open-sourced implementation is fully parallelized using *OpenMP* [47]. These improvements make *ChipletPart* readily adaptable for any chiplet-based design flow. We present a detailed runtime breakdown of *ChipletPart* in Table 8. Approximately 67% of the execution time is spent on floorplanning, while 28% is spent on chiplet cost evaluation. Further details on *ChipletPart*'s runtime are provided in Table 7.

Scalability. For large designs such as WS_4, our runtime remains under two hours. To improve scalability further, hierarchical decomposition techniques that exploit structural regularity in the netlist, along with aggressive solution pruning and potential usage of surrogate models for faster evaluation, can be employed.

6.4 Exploring Bayesian Optimization

Bayesian optimization (BO) is a powerful framework for optimizing expensive, black-box functions, particularly when gradients are unavailable and each function evaluation incurs significant computational overhead [57]. Unlike exhaustive or random search strategies, BO constructs a probabilistic *surrogate model* of the objective function and uses an *acquisition function* [58] to balance exploration and exploitation. These properties make BO especially suitable for problems with complex constraints, cost landscape, and costly evaluations. To provide another quality-runtime tradeoff point, we have also explored the potential use of BO for chiplet partitioning. We jointly optimize (i) the assignment of IP blocks to chiplets, and (ii) the selection of technology nodes per chiplet to minimize total system cost. Directly optimizing the partitioning function $\phi : \mathcal{S} \rightarrow C$ is infeasible due to the combinatorial explosion in the search space: for k chiplets and $|\mathcal{S}|$ blocks, there are $k^{|\mathcal{S}|}$ possible assignments. Such a high-dimensional, discrete space is ill-suited to BO, which relies on continuous, low-dimensional representations [59]. To address this, we use spectral embeddings to derive a continuous relaxation of the partitioning problem [60]. We construct a normalized Laplacian matrix [61]

Table 9. Comparison between *ChipletPart-GA* and *ChipletPart-BO*. *ChipletPart-GA* denotes *ChipletPart* using GA as the optimizer. *ChipletPart-BO* denotes *ChipletPart* using BO as the optimizer. Runtimes reported are the CPU time.

Bench.	**ChipletPart-GA**					**ChipletPart-BO**				
	Cost	#7nm	#10nm	#14nm	Runtime (s)	Cost	#7nm	#10nm	#14nm	Runtime (s)
WS_1	41.7	2	0	4	401	41.7	1	0	4	848
WS_2	**87.8**	1	0	7	1421	97.1	0	3	5	4262
WS_3	254.4	3	0	5	3240	**251.4**	4	0	4	9839
WS_4	656.2	7	1	0	5624	**640.2**	6	1	1	12149
MP	5.7	1	0	0	180	5.7	1	0	0	350
TC_1	**39.8**	3	0	2	160	42.1	4	1	1	1137
TC_2	**39.3**	4	0	2	89	40.2	5	1	2	959
EPYC	**65.1**	3	0	2	349	67.8	4	4	0	1263
GA100	31.6	3	1	0	493	31.6	3	1	0	1685

from the netlist connectivity graph and compute its d nontrivial eigenvectors. These eigenvectors define a mapping:

$$\Phi_{\text{embed}} : \mathcal{S} \to \mathbb{R}^d, \quad s \mapsto \mathbf{e}_s$$

where $s \in \mathcal{S}$ is an IP block, and $\mathbf{e}_s \in \mathbb{R}^d$ is its coordinate in the d-dimensional spectral embedding space. Blocks that are tightly coupled (i.e., frequently communicate or share many nets) are embedded closer together. This transformation preserves the structural information of the netlist in a form more amenable to clustering [5]. Rather than directly optimizing ϕ, we allow BO to propose a set of k centroids $\{\mathbf{c}_1, \ldots, \mathbf{c}_k\}$ in $\mathbb{R}^d$. Each block is then assigned to its nearest centroid:

$$\phi(s) = \arg \min_{j \in \{1,\ldots,k\}} \|\mathbf{e}_s - \mathbf{c}_j\|_2 \tag{4}$$

This transforms the partitioning problem from a high-dimensional discrete search into a continuous optimization over $k \cdot d$ real-valued variables. Moreover, this centroid-based assignment strategy naturally produces geometrically coherent partitions and enables integration with downstream refinement algorithms such as FM or KL.

In addition to partitioning, the BO framework must also determine the technology assignment $\omega(c) \in \mathcal{T}$ for each chiplet $c \in C$. Technology assignments are categorical variables, which are challenging to surrogates due to their non-ordinal nature. To mitigate this, we encode each technology assignment using a one-hot vector:

$$\mathbf{t}_i \in \{0,1\}^m, \quad \text{where } m = |\mathcal{T}|, \quad \sum_{j=1}^{m} \mathbf{t}_i[j] = 1$$

Here, $\mathbf{t}_i$ represents the technology assigned to chiplet c_i. This encoding allows the BO surrogate model to treat technology assignment as a structured subspace within a continuous optimization landscape. Each candidate solution evaluated by BO is represented as a vector x:

- The number of partitions k (integer-valued).
- k centroids in d-dimensional spectral space: $\mathbf{c}_1, \ldots, \mathbf{c}_k$.
- One-hot technology encodings $\mathbf{t}_1, \ldots, \mathbf{t}_k$.

Formally, $x = [k, \mathbf{c}_1, \ldots, \mathbf{c}_k, \mathbf{t}_1, \ldots, \mathbf{t}_k]$ and the dimensionality of x is: $\dim(x) = 1 + k \cdot d + k \cdot m$. The BO engine constructs a surrogate model $\hat{\Phi}(x)$ to approximate the true cost $\Phi(x)$ and uses an acquisition function $\alpha(x)$ to guide exploration. We use a Gaussian Process (GP) [62] surrogate for small datasets (< 50 IP blocks) and Random Forest [63] for larger datasets. Our acquisition function comprises Expected Improvement [58] and Upper Confidence Bound [58] metrics. Once BO picks a *candidate* x, it undergoes the following evaluation pipeline:

(1) *Partition decoding*: Assign each block $s \in \mathcal{S}$ to its nearest centroid to obtain partition ϕ.
(2) *FM refinement*: Improve the partition using *ChipletPart*'s FM refinement.
(3) *Technology decoding*: Map each one-hot vector $\mathbf{t}_i$ to its corresponding technology node $\omega(c_i)$.
(4) *Cost evaluation*: Call the *ChipletPart*'s cost model to evaluate the cost $\Phi(\phi, \omega)$, and floorplan checks.
(5) *Feasibility check*: If the solution is infeasible, i.e., does not satisfy the reach constraints, return a large penalty value $M \gg 1$: $\Phi(\phi, \omega) = M$. This ensures that the BO explores the landscape where the solutions are always floorplan-feasible.

The result is fed back to update the surrogate model, and the process is repeated until a termination criterion is met (we use a fixed limit of 200 iterations). We compare our implemented BO framework with our *ChipletPart*'s GA framework. Table 9 presents a cost and runtime comparison of the two optimizers — *ChipletPart-GA* and *ChipletPart-BO*. The results show that *ChipletPart-BO* can produce lower-cost solutions on the larger designs, WS_3 and WS_4, where it achieves cost reductions of up to 5.3% and 2.4%, respectively. However, the benefits come at a significant computational cost. On average, *ChipletPart-BO* incurs a runtime overhead of 2×–4× compared to *ChipletPart-GA*, due to its expensive surrogate model updates. In small-to-medium benchmarks such as MP, TC_2, and $GA100$, both optimizers achieve comparable costs, but BO remains considerably slower. Thus, GA and BO provide a quality-runtime trade-off.

BO vs. GA trade-offs. Our results show that *ChipletPart-GA* provides the best overall quality/runtime tradeoff for chiplet partitioning: it converges quickly, is robust across benchmarks, and integrates naturally into a broader CAD flow. In contrast, *ChipletPart-BO* incurs significantly higher runtime due to surrogate-model construction and acquisition-function optimization, but can yield modestly better solutions on benchmarks whose spectral embeddings exhibit a smoother cost landscape (e.g., WS_3, WS_4). We recommend *ChipletPart-GA* for routine use or time-constrained flows, while *ChipletPart-BO* can potentially serve as a compute-intensive alternative that may improve quality on select, more challenging testcases.

7 ADDITIONAL CASE STUDIES

We investigate how different technology parameters affect *ChipletPart*'s solutions. To accommodate an arbitrarily complex cost model, *ChipletPart* employs a black-box partitioning method rather than a mathematical programming approach. This design choice enables a broader exploration of the parameter space compared to other partitioners. To illustrate these benefits, we present case studies that examine effects of altering the chiplet cost model parameters.

I/O type and reach. Chiplet systems often use reduced-size I/O cells that trade driver strength and electrostatic discharge (ESD) protection for a smaller form factor. However, limited I/O reach makes it challenging to generate feasible solutions. A 2 mm reach may only allow direct connections between adjacent chiplets, while a larger reach (e.g., 20 mm) increases flexibility but also adds to system cost by increasing chiplet area. To explore this, we: (i) analyze the impact of I/O reach on

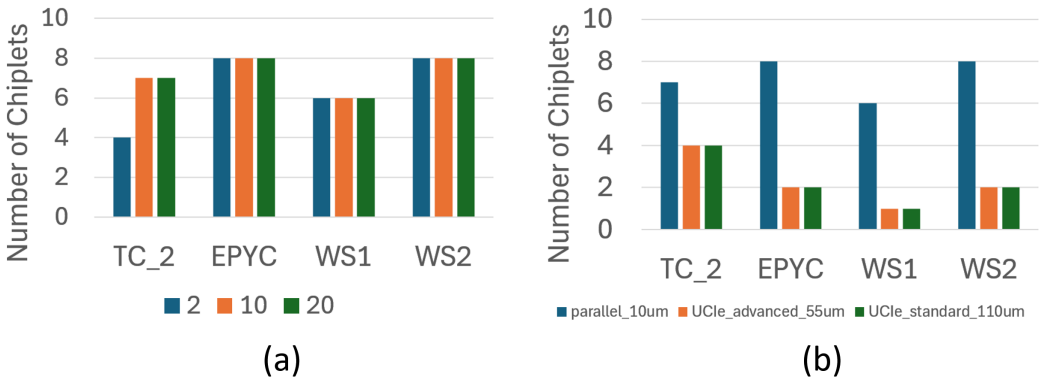

Fig. 14. (a) Reach limit vs. #chiplets. The partitioning solutions are generated with *ChipletPart* using reach values of 2 mm, 10 mm, and 20 mm. (b) I/O type vs. #chiplets.

chiplet partitioning for a *parallel* I/O testcase; and (ii) compare the *parallel* I/O testcase with UCIe *advanced* and UCIe *standard* definitions [31], which have different cell sizes and reach. The parallel I/O has 2 mm reach, while the UCIe cases have 10 mm reach.

Figure 14(a) shows that increasing reach enables a greater number of chiplets in the partitioning solution, while a short 2 mm reach forces more blocks to remain together in fewer chiplets. As shown in Figure 4, longer I/O reach values enable connections that are impossible under shorter reach limits. A reach of 2 mm effectively restricts connectivity to nearest neighbors, which forces more blocks to remain colocated and increases overall cost. This effect is clearly illustrated in the TC_2 testcase: with a 2 mm reach, the resulting solution has a cost of 46.3, whereas increasing the reach to 10 mm produces a lower-cost solution (cost 43.4), corresponding to a 6.3% reduction. Figure 14(b) compares the same *parallel* I/O testcase with UCIe interfaces, showing that UCIe's large I/O cells favor fewer chiplets, while the parallel interface with smaller I/Os and bump pitches enables finer partitioning. For example, this effect is evident in the *EPYC* testcase. We verified the impact of I/O type and bump pitch by evaluating the cost of a fixed chiplet partitioning solution, the *parallel_10um* solution, under the other I/O configurations. When evaluated with the *parallel_10um* I/O configuration, the solution has a cost of 73.1, whereas evaluating the same partitioning solution under the *UCIe_advanced_55um* and *UCIe_standard_110um* configurations increases the cost to 74.1 and 74.5, respectively. Because this additional cost arises solely from the larger I/O cell sizes and pitches, the solutions optimized for *UCIe_advanced_55um* and *UCIe_standard_110um* reduce the total inter-chiplet connection bandwidth by merging blocks into fewer chiplets, thereby avoiding the higher I/O cost.

Mixed cost/power objective function. *ChipletPart* can simultaneously optimize cost and power using a weighted sum of chiplet cost (Equation 1) and power as its objective function. Table 10 shows results for various cost and power coefficient combinations. For this experiment, we use the WS_1 and TC_1 testcases, with all chiplets implemented in 7 nm technology. As the power coefficient increases, the solutions favor fewer chiplets since large I/O drivers for inter-chiplet connections incur higher power consumption.

8 CONCLUSION

We have proposed *ChipletPart*, a novel chiplet partitioning framework designed to tackle the challenges of partitioning large, 2.5D heterogeneously integrated systems into chiplets. *ChipletPart* leverages an advanced cost model, is power-aware, and uses a genetic algorithm to simultaneously assign

Table 10. Effects of different cost and power coefficient combinations. Cost_N and Power_N refer to normalized cost and power, respectively.

Coefficients		WS_1			TC_1		
Cost	**Power**	$\|C\|$	$\textbf{Cost}_N$	$\textbf{Power}_N$	$\|C\|$	$\textbf{Cost}_N$	$\textbf{Power}_N$
1	0	6	**1.00**	**1.00**	6	**1.00**	**1.00**
0.75	0.25	6	1.02	0.97	4	1.03	0.97
0.5	0.5	4	1.06	0.95	2	1.04	0.95
0.25	0.75	4	1.07	0.93	2	1.07	0.93
0	1	2	1.10	0.92	1	1.08	0.91

technologies to chiplets, all while consistently returning I/O-feasible solutions that honor the driving reach of inter-chiplet I/O drivers. Experimental results demonstrate that *ChipletPart* outperforms classical netlist partitioners, *Floorplet* [7], and human baselines. Additional studies illuminate dependencies of partitioning outcomes on the underlying packaging and I/O technology. We have also explored Bayesian optimization as an alternative search strategy. Although Bayesian optimization can sometimes generate higher-quality solutions than the genetic algorithm, it incurs significantly higher runtime overhead.

While *ChipletPart* is designed as a cost-driven partitioning framework, its modular, black-box optimization approach promotes future integration of more complex objective functions that consider power, performance and thermal factors. Extending the current objective function to include additional factors remains a direction for future work. Considering detailed I/O planning is another direction for future work. Detailed I/O planning that supports pass-through connections, multiple I/O types on a die, "gas station" buffers [64], buffer chiplets, or other similar architectural considerations will bring significantly increased problem complexity. Evaluation with routing-congested designs is another topic of interest. We also aim to enhance *ChipletPart* by identifying repeated structures in the netlist, a capability that could significantly improve scalability for larger designs. Beyond 2.5D, adapting *ChipletPart* to 3D-stacked technologies, where vertical interconnects, TSV constraints, and tier locality introduce new partitioning objectives, is an opportunity for future work. Incorporating detailed thermal and power delivery models can potentially enable co-optimization for reliability, temperature, and system-level performance. Finally, we are exploring the use of alternate optimizers, such as Integer Linear Programming (ILP) and Reinforcement Learning (RL). We also plan to integrate *ChipletPart* with OpenROAD [65] to promote engagement in benchmarking and further development.

ACKNOWLEDGMENTS

This work was supported in part by Samsung AI Center, Intel Corporation, DARPA/SRC CHIMES JUMP 2.0 center, NSF, and the CDEN center.

REFERENCES

[1] W. Gomes, A. Koker, P. Stover, D. Ingerly, S. Siers, S. Venkataraman, C. Pelto, T. Shah, A. Rao, F. O'Mahony, E. Karl, L. Cheney, I. Rajwani, H. Jain, R. Cortez, A. Chandrasekhar, B. Kanthi, and R. Koduri, "Ponte vecchio: A multi-tile 3D stacked processor for exascale computing," in *Proc. ISSCC*, 2022, pp. 42–44.

[2] A. Sangiovanni-Vincentelli, Z. Liang, Z. Zhou, and J. Zhang, "Automated design of chiplets," in *Proc. ISPD*, 2023, pp. 1–8.

[3] A. Graening, S. Pal, and P. Gupta, "Chiplets: How small is too small?" in *Proc. DAC*, 2023, pp. 1–6.

[4] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Applications in vlsi domain," *IEEE Trans. on VLSI*, vol. 7, no. 1, pp. 69–79, 1999.

[5] I. Bustany, A. B. Kahng, I. Koutis, B. Pramanik, and Z. Wang, "Specpart: A supervised spectral framework for hypergraph partitioning solution improvement," in *Proc. ICCAD*, 2022, pp. 1–9.
[6] I. Bustany, G. Gasparyan, A. B. Kahng, I. Koutis, B. Pramanik, and Z. Wang, "An open-source constraints-driven general partitioning multi-tool for VLSI physical design," in *Proc. ICCAD*, 2023, pp. 1–9.
[7] S. Chen, S. Li, Z. Zhuang, S. Zheng, Z. Liang, T.-Y. Ho, B. Yu, and A. L. Sangiovanni-Vincentelli, "Floorplet: Performance-aware floorplan framework for chiplet integration," *IEEE Trans. on CAD*, vol. 43, no. 6, pp. 1638–1649, 2024.
[8] F. Li, Y. Wang, Y. Wang, M. Wang, Y. Han, H. Li, and X. Li, "Chipletizer: Repartitioning SoCs for cost-effective chiplet integration," in *Proc. ASP-DAC*, 2024, pp. 58–64.
[9] A. Graening, J. Talukdar, S. Pal, K. Chakrabarty, and P. Gupta, "CATCH: A cost analysis tool for co-optimization of chiplet-based heterogeneous systems," arXiv preprint arXiv:2503.15753, 2025.
[10] A. Graening, S. Pal, and P. Gupta, "CATCH chiplet cost model," https://github.com/nanocad-lab/CATCH, 2025.
[11] "ChipletPart repository," https://github.com/ABKGroup/ChipletPart/blob/main/README.md, 2025.
[12] Y. Feng and K. Ma, "Chiplet actuary: A quantitative cost model and multi-chiplet architecture exploration," in *Proc. DAC*, 2022, pp. 121–126.
[13] Z. Zhuang, B. Yu, K.-Y. Chao, and T.-Y. Ho, "Multi-package co-design for chiplet integration," in *Proc. ICCAD*, 2022, pp. 1–9.
[14] A. Graening, D. A. Patel, G. Sisto, E. Lenormand, M. Perumkunnil, N. Pantano, V. B. Kumar, P. Gupta, and A. Mallik, "Cost-performance co-optimization for the chiplet era," in *Proc. EPTC*, 2024, pp. 40–45.
[15] M. A. Kabir and Y. Peng, "Chiplet-package co-design for 2.5D systems using standard ASIC CAD tools," in *Proc. ASP-DAC*, 2020, pp. 351–356.
[16] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proc. SC*, 1996, pp. 35–35.
[17] C. Roman-Vicharra, Y. Chen, and J. Hu, "PPAC driven multi-die and multi-technology floorplanning," arXiv preprint arXiv:2502.10932, 2025.
[18] Y.-K. Ho and Y.-W. Chang, "Multiple chip planning for chip-interposer codesign," in *Proc. DAC*, 2013, pp. 1–6.
[19] C.-C. Lee and Y.-W. Chang, "Floorplanning for embedded multi-die interconnect bridge packages," in *Proc. ICCAD*, 2023, pp. 1–8.
[20] H.-W. Chiou, J.-H. Jiang, Y.-T. Chang, Y.-M. Lee, and C.-W. Pan, "Chiplet placement for 2.5D IC with sequence pair based tree and thermal consideration," in *Proc. ASP-DAC*, 2023, pp. 7–12.
[21] W.-H. Liu, M.-S. Chang, and T.-C. Wang, "Floorplanning and signal assignment for silicon interposer-based 3D ICs," in *Proc. DAC*, 2014, pp. 1–6.
[22] D. Aldous and U. Vazirani, "Go with the winners algorithms," in *Proc. FOCS*, 1994, pp. 492–501.
[23] B. L. Miller and D. E. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex Systems*, vol. 9, no. 3, pp. 193–212, 1995.
[24] G. Syswerda, "Uniform crossover in genetic algorithms," in *Proc. ICGA*, vol. 3, 1989, pp. 2–9.
[25] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.
[26] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74–81, 2017.
[27] K. Heyman, "SRAM scaling issues, and what comes next," 2024. [Online]. Available: https://semiengineering.com/sram-scaling-issues-and-what-comes-next/
[28] M. Lapedus, "Big trouble at 3nm," 2018. [Online]. Available: https://semiengineering.com/big-trouble-at-3nm/
[29] D. Patel, "The dark side of the semiconductor design renaissance — fixed costs soaring due to photomask sets, verification, and validation," 2023. [Online]. Available: https://www.semianalysis.com/p/the-dark-side-of-the-semiconductor
[30] A. Shilov, "TSMC's estimated wafer prices revealed: 300mm wafer at 5nm is nearly $17,000," 2020. [Online]. Available: https://www.tomshardware.com/news/tsmcs-wafer-prices-revealed-300mm-wafer-at-5nm-is-nearly-dollar17000
[31] "UCIe specification 1.0." [Online]. Available: https://www.uciexpress.org/team-3
[32] S. Pal, J. Liu, I. Alam, N. Cebry, H. Suhail, S. Bu, S. S. Iyer, S. Pamarti, R. Kumar, and P. Gupta, "Designing a 2048-chiplet, 14336-core waferscale processor," in *Proc. DAC*, 2021, pp. 1183–1188.
[33] S. Pal, I. Alam, K. Sahoo, H. Suhail, R. Kumar, S. Pamarti, P. Gupta, and S. S. Iyer, "I/O architecture, substrate design, and bonding process for a heterogeneous dielet-assembly based waferscale processor," in *Proc. ECTC*, 2021, pp. 298–303.
[34] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. DAC*, 1982, pp. 175–181.
[35] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[36] U. V. Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, pp. 395–416, 2007.
[37] A. Likas, N. Vlassis, and J. J. Verbeek, "The global k-means clustering algorithm," *Pattern Recognition*, vol. 36, no. 2, pp. 451–461, 2003.
[38] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable k-means++," *arXiv preprint arXiv:1203.6402*, 2012.
[39] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, "Graph edge partitioning via neighborhood heuristic," in *Proc. SIGKDD*, 2017, pp. 605–614.
[40] "METIS repository," https://github.com/KarypisLab/METIS, 2024.
[41] H. Abdi, "Z-scores," *Encyclopedia of Measurement and Statistics*, vol. 3, pp. 1055–1058, 2007.
[42] "KL refinement (KLRefiner.cpp)," https://github.com/ABKGroup/ChipletPart/blob/main/src/KLRefiner.cpp, 2025.
[43] "Reach violation calculation (floorplan.cpp)," https://github.com/ABKGroup/ChipletPart/blob/693f2fc/src/floorplan.cpp#L576, 2025.
[44] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Trans. on CAD*, vol. 15, no. 12, pp. 1518–1524, 1996.
[45] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
[46] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on b*-tree and fast simulated annealing," *IEEE Trans. on CAD*, vol. 25, no. 4, pp. 637–650, 2006.
[47] "OpenMP library," https://www.openmp.org/, 2021.
[48] "Eigen v.3.4.0," https://gitlab.com/libeigen/eigen/-/releases/3.4.0, 2021.
[49] "Boost library v.1.87.0," https://www.boost.org/releases/1.87.0/, 2024.
[50] B. Landman and R. Russo, "On a pin versus block relationship for partitions of logic graphs," *IEEE Trans. on Computers*, vol. C-20, no. 12, pp. 1469–1479, 1971.
[51] P. Singh and D. Landis, "Optimal chip sizing for multi-chip modules," *IEEE Trans. on CPMT*, vol. 17, no. 3, pp. 369–375, 1994.
[52] M. Cavalcante, S. Riedel, A. Pullini, and L. Benini, "Mempool: A shared-L1 memory many-core cluster with a low-latency interconnect," in *Proc. DATE*, 2021, pp. 701–706.
[53] W. Liao, C. Chiang, W. Wu, C. H. Fan, S. Chiu, C. Chiu, T. Chen, C. Hsieh, H. Chen, H. Lo, L. Huang, T. Wu, W. Chiou, S. Hou, S. Jeng, and D. Yu, "A high-performance low-cost chip-on-wafer package with sub-µm pitch Cu RDL," in *Symp. VTDTP*, 2014, pp. 1–2.
[54] M. Hempel and J. Redford, "Personal communication," 2024.
[55] "AMD EPYC 7282." [Online]. Available: https://www.techpowerup.com/cpu-specs/epyc-7282.c2255
[56] "NVIDIA GA100." [Online]. Available: https://www.techpowerup.com/gpu-specs/nvidia-ga100.g931
[57] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. D. Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proc. of the IEEE*, vol. 104, no. 1, pp. 148–175, 2015.
[58] W. Gan, Z. Ji, and Y. Liang, "Acquisition functions in bayesian optimization," in *Intl. Conf. on BASE*. IEEE, 2021, pp. 129–135.
[59] L. Papenmeier, M. Poloczek, and L. Nardi, "Understanding high-dimensional bayesian optimization," *arXiv preprint arXiv:2502.09198*, 2025.
[60] F. R. Chung, *Spectral graph theory*. American Mathematical Society, 1997, vol. 92.
[61] R. Merris, "Laplacian matrices of graphs: A survey," *Linear Algebra and its Applications*, vol. 197, pp. 143–176, 1994.
[62] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.
[63] W. Zhang, C. Wu, H. Zhong, Y. Li, and L. Wang, "Prediction of undrained shear strength using extreme gradient boosting and random forest based on bayesian optimization," *Geoscience Frontiers*, vol. 12, no. 1, pp. 469–477, 2021.
[64] A. Coskun, F. Eris, A. Joshi, A. B. Kahng, Y. Ma, A. Narayan, and V. Srinivas, "Cross-layer co-optimization of network design and chiplet placement in 2.5-D systems," *IEEE Trans on CAD*, vol. 39, no. 12, pp. 5183–5196, 2020.
[65] "The OpenROAD project (commit hash: db9f42f)," https://github.com/The-OpenROAD-Project/OpenROAD.