

Fast, Robust, and Transferable Prediction for Hardware Logic Synthesis

Ceyu Xu
ceyu.xu@duke.edu
Duke University
Durham, North Carolina, USA

Tianshu Wang
tianshu.wang@duke.edu
Duke University
Durham, North Carolina, USA

Pragya Sharma
pragya.sharma@duke.edu
Duke University
Durham, North Carolina, USA

Lisa Wu Wills
lisa@cs.duke.edu
Duke University
Durham, North Carolina, USA

ABSTRACT

The increasing complexity of computer chips and the slow logic synthesis process have become major bottlenecks in the hardware design process, also hindering the ability of hardware generators to make informed design decisions while considering hardware costs. While various models have been proposed to predict physical characteristics of hardware designs, they often suffer from limited domain adaptability and open-source hardware design data scarcity.

In this paper, we present SNS v2, a fast, robust, and transferable hardware synthesis predictor based on deep learning models. Inspired by modern natural language processing models, SNS v2 adopts a three-phase training approach encompassing pre-training, fine-tuning, and domain adaptation, enabling it to leverage more abundant unlabeled and off-domain training data. Additionally, we propose a novel contrastive learning approach based on circuit equivalence to enhance model robustness. Our experiments demonstrate that SNS v2 achieves two to three orders of magnitude faster speed compared to conventional EDA tools, while maintaining state-of-the-art prediction accuracy. We also show that SNS v2 can be seamlessly integrated into hardware generator frameworks for real-time cost estimation, resulting in higher quality design recommendations in a significantly reduced time frame.

CCS CONCEPTS

• **Hardware** → **Integrated circuits; High-level and register-transfer level synthesis; Logic synthesis; Computing methodologies** → **Machine learning.**

KEYWORDS

Integrated Circuits, RTL-level Synthesis, Logic Synthesis Prediction, Neural Networks

ACM Reference Format:

Ceyu Xu, Pragya Sharma, Tianshu Wang, and Lisa Wu Wills. 2023. Fast, Robust, and Transferable Prediction for Hardware Logic Synthesis. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*, October 28–November 1, 2023, Toronto, ON, Canada. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3613424.3623794>

1 INTRODUCTION

The exponential growth in digital circuits (e.g., more transistors per area, bigger chips, better manufacturing yield) has enabled more complex and more capable hardware designs in recent decades. Revolutionary advancements in the last two years in deep learning, such as diffusion models used for image generation (e.g., DALL-E2 [22]) and large language models with trillions of parameters used for text generation (e.g., GPT-4 [21], LLaMA [28]), posed unprecedented computational challenges. These computational challenges necessitate even more sophisticated and versatile hardware designs, resulting in an accelerated increase in hardware design complexity. To evaluate and fabricate these ever-complex hardware designs, much longer logic synthesis runtime is required to accurately model hardware cost (e.g., power, area, and timing) and adhere to intricate design rules and constraints (especially for sub-14nm technology libraries). Further, optimizing for multiple targets such as minimizing power consumption in addition to maximizing performance calls for the consideration of a much wider array of design choices, resulting in hardware synthesis being a more time-consuming and resource-intensive task.

In recent years, advanced machine learning techniques have been introduced to predict the synthesis results of hardware designs [15, 17, 29, 37, 40] in order to alleviate the need for running hundreds of designs through traditional, time-consuming synthesis tool chains (e.g., Synopsys Design Compiler [26]). However, these existing works have several drawbacks. First, they often provide predictions of one technology library only and they rely on a large dataset to train the model [29, 37, 40]. For example, to predict synthesis results for a 32nm technology node, the model is trained with hardware synthesis “ground-truth” results generated via Synopsys DC using a 32nm technology library. To predict synthesis results for a different technology node, such as 14nm or 7nm, the model will have to be re-trained on 14nm or 7nm “ground-truth” synthesis results.

Second, prior works may only work for a specific domain of hardware designs rather than any arbitrary hardware designs. For

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MICRO '23, October 28–November 1, 2023, Toronto, ON, Canada

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0329-4/23/10...\$15.00

<https://doi.org/10.1145/3613424.3623794>

example, MAESTRO [17] framework only predicts the cost and performance of neural network accelerators that follow their provided taxonomy, and is unable to predict designs outside of the domain; Aladdin [15] is only able to predict the designs that adapt to a pre-defined SoC accelerator architecture. Third, models using a graph neural network for prediction cannot scale to large enough designs such as D-SAGE[29] and GRANNITE [40]. GRANNITE achieves 2045× speedup when predicting the power consumption of a small adder pipeline while the speedup drops to 19× when predicting a mini-RISC-V core.

In this work, we present SNS v2, a novel *transferable* deep-learning-based logic synthesis predictor that is capable of predicting synthesis results of *arbitrary hardware designs on multiple technology nodes*. SNS v2 improves the accuracy and robustness of the original SNS model; in addition, it provides transferability across technology libraries, which was not possible with the original SNS model. SNS v2 employs the three-phase training approach used widely in generative language models as to not have to retrain the model for every technology node or operating condition with large datasets and expensive re-training. In addition, SNS v2 uses a two-level hierarchical graph neural network model by decomposing large designs into smaller designs and training with high-quality “ground-truth” synthesis results of smaller designs to ensure that the model is scalable and can predict very large designs. SNS v2 can predict synthesis results with significant speedup vs. traditional synthesis toolchain while providing state-of-the-art accuracy and robustness. SNS v2 focuses on predicting the logic synthesis results (which only includes the area, power and timing delay) of RTL designs, and thus detailed timing or runtime power reports, modeling of the scratchpads or SRAMs, and predicting post-placement-and-route results, are beyond the scope of this work.

We make the following contributions:

- We introduce an exceptionally rapid logic synthesis predictor, SNS v2, capable of predicting synthesis results for a diverse set hardware designs two to three orders of magnitude faster than conventional synthesis toolchains. Our model demonstrates state-of-the-art accuracy, robustness, and generalizability in comparison to existing logic synthesis predictors.
- We propose an innovative three-phase training methodology that takes advantage of circuit functional equivalences, facilitating the development of a highly generalizable logic synthesis predictor using extensive unlabeled hardware design data.
- We present a case study illustrating the advantages of fast cost estimation for complex hardware generators and the seamless integration of SNS v2 into a hardware generator to enable such rapid estimations. The case study demonstrates that, with our model’s guidance, superior quality design recommendations can be made in a significantly reduced time frame.

2 MOTIVATION AND BACKGROUND

Machine learning has emerged as a powerful tool for addressing complex problems in various domains including computer architecture. Thus, researchers have developed diverse machine learning

techniques to tackle different aspects of hardware design, optimization, and analysis.

One prevalent approach in this area is the application of Graph Neural Networks (GNNs) for hardware designs. GNNs are particularly well-suited for analyzing and learning from graph-structured data, which is a natural fit for representing hardware components and their interconnections. By leveraging GNNs, researchers have made significant advancements in various tasks such as placement and routing prediction [34], logic synthesis result prediction [37], power consumption prediction [36], and performance estimation [33].

While GNNs have shown great promises in hardware design, their use still presents us with two inherent challenges. First, with the increasing complexity and size of modern hardware designs, GNNs struggle with high computational overhead and memory consumption. Second, over-smoothing, when the graph node representations tend to become indistinguishable as the number of neural network layers increases, can be particularly detrimental for hardware designs, since it can lead to the loss of local information.

To address these challenges, we propose the use of hierarchical GNNs (HGNNs), since they not only scale much better compared to typical message-passing GNNs, but also preserve local information during training and inference.

2.1 Three-phase Training for Transfer Learning and Data-Efficient Learning

Machine learning models are highly dependent on data, and limited data availability can lead to poor learning performance. To address this issue, numerous models and techniques have been proposed. One of these techniques is the three-phase training approach, which is especially effective for data-efficient learning. This approach, inspired by the success of generative language models, is comprised of three stages: pre-training, fine-tuning, and domain-specific adaptation. (Note: the word *domain* used in the machine learning context, refers to a *task domain* such as a semantic analysis task domain, unlike application domain used in the computer architecture context.) The motivation for this approach is that well-labeled data for a specific domain is often scarce, while labeled data in a broader domain is relatively more abundant, and unlabeled data is typically abundant. Because of this, the three-phase training method leverages the abundant unlabeled data and off-domain labeled data to improve the model’s performance on the scarce in-domain labeled data. (Note: using the previous example to explain *domain*, a pre-trained language model may have an *in-domain* task that is a question-answering task. An *off-domain* task in this context would correspond to another language task domain such as semantic analysis.)

An example of the three-phase training approach is the BERT model [5], which was designed for natural language processing. First, the token (word) embedding of BERT is pre-trained on a vast corpus of unlabeled text (e.g., all written content on the internet). Next, the entire BERT model is fine-tuned on a large corpus of labeled text obtained from various domains. Finally, the BERT model can be further fine-tuned on a specific domain, such as sentiment analysis, using only a small amount of labeled data to achieve higher performance.

Data scarcity has been a significant challenge for previous works [29, 34, 37] that apply deep learning models to hardware designs. In this work, we address this issue by using the three-phase training approach to predict the synthesis results of hardware designs. First, we pre-train our model using a large amount of unlabeled designs. Next, we fine-tune the model in a supervised manner, leveraging feedback from synthesis tools. Finally, we demonstrate that our model can transfer to another technology library, even with a test set as small as one sample.

2.2 Contrastive Learning for Robustness of the Model

The primary benefit of the pre-training phase lies in its ability to enable the model to learn circuit design representations without relying on labeled data. However, what forms the foundation for pre-training the model remains a critical question. Popular pre-training methods, such as Word2Vec [19], Node2Vec [9], and masked language models [5], rely on the assumption that similar words, nodes, and tokens should be represented by similar vectors. But in the context of hardware designs, what constitutes similarity remains unclear.

In this work, we investigate the notion of similarity in the context of hardware designs by analyzing the limitations of existing works. Previous research [29, 34, 35, 37] predicts hardware costs using some type of intermediate representation (IR) of the hardware designs. The machine learning models then forecast the physical properties of the hardware design based on the IR input. However, this approach has a fundamental flaw: two distinct circuit representations may depict the same functionality and identical circuit implementation. For instance, $a + b$ and $a + b + 1 - 1$ represent equivalent computational circuits, but they have different IRs and appear to be different. In fact, compilers and EDA (electronic design automation) tools utilize these legal transformations and equivalences of circuits for optimization.

We propose a contrastive-learning-based, pre-training method that assists the model in learning circuit equivalences. By pre-training the model in this way, we found that it not only improved the model’s ability to infer the equivalence of hardware circuits but also substantially enhanced the overall accuracy of downstream tasks such as predicting synthesis results. More details on contrastive learning is described in Section 3.4.

3 MODEL

The three-phase training process is well-regarded for enhancing a model’s performance, especially with limited data, which is crucial for predicting hardware synthesis outcomes. In this study, we propose a novel approach by applying the three-phase training process to our synthesis predictor, SNS v2. This approach is similar to NLP model training but incorporates domain knowledge of RTL circuits.

Figure 1 provides an overview of SNS v2. SNS v2 takes a hardware design in Verilog, SystemVerilog, or FIRRTL [12] format as input and utilizes our circuit graph compiler ① to convert the input hardware description language (HDL) into a graph intermediate representation we call *circuitIR*, shown in Figure 1(a). FIRRTL, or Flexible Intermediate Representation for RTL, is an intermediate representation for digital circuits, which we use in our model. It is

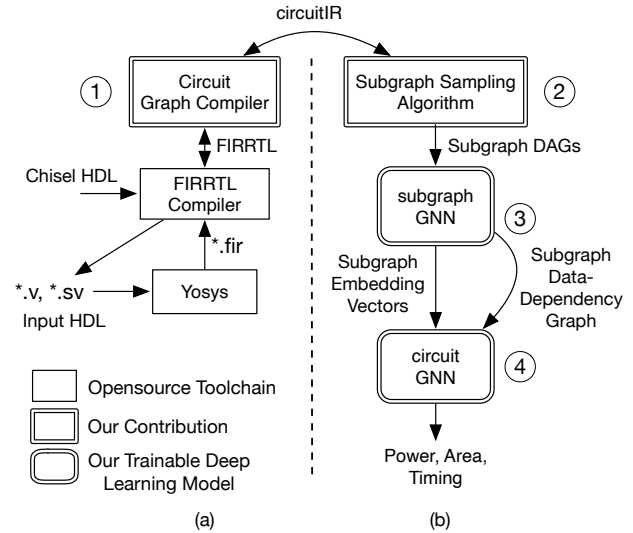


Figure 1: Overview of SNS v2.

part of the Chisel HDL compilation toolchain [3]. FIRRTL represents the circuit immediately after Chisel’s elaboration and is designed to appear identical to the Chisel HDL once all meta-programming has been executed. FIRRTL is not only tied to Chisel, as other HDLs written in various languages can target FIRRTL and reuse the majority of the compilation toolchain.

Once we obtain the intermediate representation, we sample subgraphs from it using our custom subgraph sampling algorithm ②, producing a set of subgraphs and a dependency graph between subgraphs as shown in Figure 1(b). We then apply a hierarchical GNN model with two levels of GNNs to predict the synthesis results. The first-level GNN is a subgraph GNN (*subgraphGNN* ③) that produces embedding vectors for the subgraphs. These embeddings are then fed into the second-level GNN, the *circuitGNN* ④, which takes the dependency graph and subgraph embeddings as input and predicts the synthesis results of the entire hardware design.

Our method’s three-phase training process is compared to typical NLP models in Figure 2. The subgraphGNN is pre-trained on unlabeled subgraphs using a contrastive method, similar to how NLP models pre-train on unlabeled data. Figure 2(a)① and 2(c)① compare the pre-training process of our method to that of a typical NLP model. During fine-tuning, both the subgraphGNN and the circuitGNN are fine-tuned using feedback from the EDA tools, rather than unlabeled, general data. Figure 2(a)② and 2(c)② compare the fine-tuning process of our method to that of a typical NLP model. Finally, as shown in Figure 2(b)③ and 2(d)③, our method and typical NLP models can be further fine-tuned on a specific task or domain, a procedure called domain adaptation. This is particularly useful for our method, as labeled datasets for specific synthesis targets can be challenging to collect.

3.1 Circuit Graph Compiler

Extracting features directly from HDLs can be challenging due to their syntax, which often provides obscure and abstract information about the implementation and functionalities, especially in code for

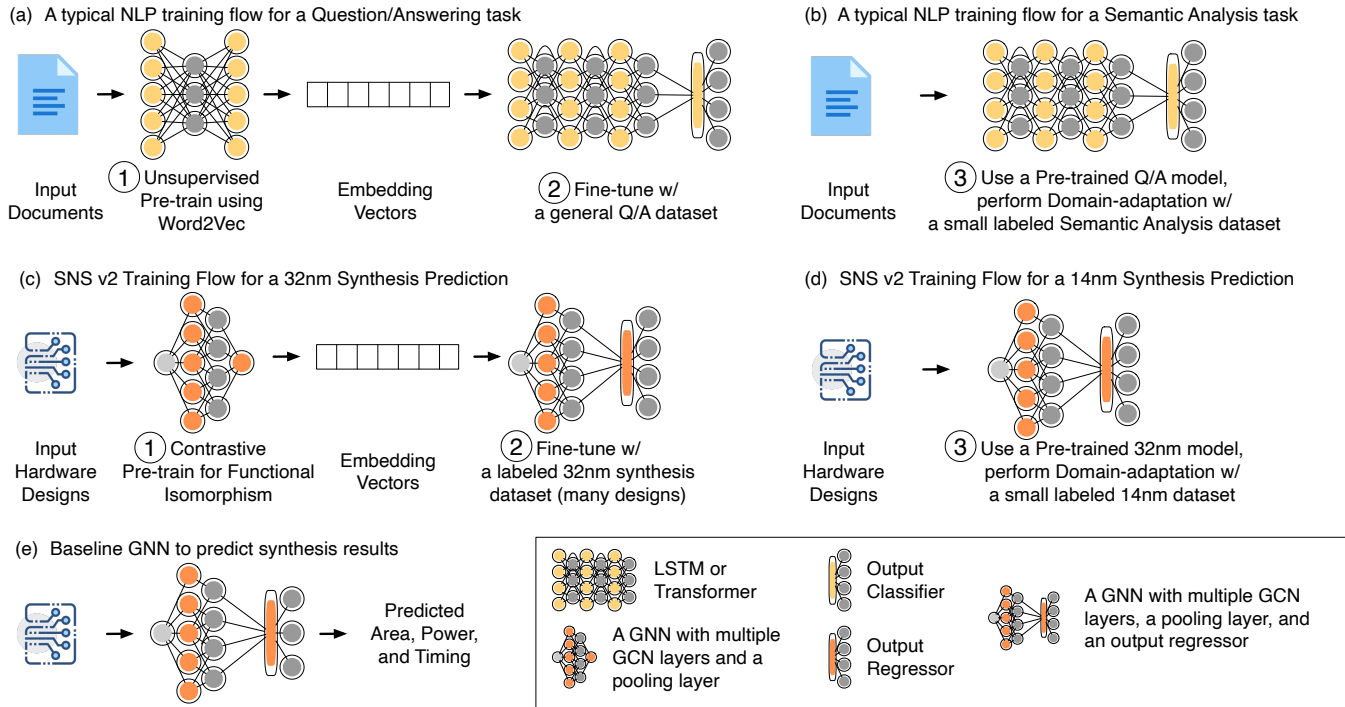


Figure 2: Analogy between SNS v2's three-phase training process and the NLP model training process.

hardware generation. To overcome this, a popular method used in previous works [29, 34, 35, 37] is adopted. This involves converting the HDLs to a graph intermediate representation that is easier for neural networks to process.

The process flow of the SNS v2 circuit graph compiler is illustrated in Figure 1. For a hardware design in Chisel HDL, the internal compiler of the Chisel framework is used to convert the HDL into the FIRRTL intermediate representation. The generated FIRRTL is then transformed into circuitIR using the SNS v2 circuit graph compiler. For a hardware design in Verilog, SystemVerilog, or VHDL, the Yosys open source synthesis suite [32] is utilized to convert the HDLs to FIRRTL. FIRRTL can then be processed by the SNS v2 circuit graph compiler to generate circuitIR.

The circuit graph compiler processes FIRRTL using two key steps. First, it flattens the hardware design into a single top-level module and creates a directed graph. Every node in the graph represents a register (reg), a wire (in, out), or an operator (e.g., add, mul, concat, bits); the edges in the graph represent the physical connections between the nodes. Second, it annotates the graph nodes with the bit-width information. For example, a node with type add and bit-width 32 (add. 32) represents a 32-bit adder.

An example hardware design for an 8-bit dot product module is shown in Figure 3. The Verilog code is listed in Figure 3(a), and the compiled circuitIR is shown pictorially in Figure 3(b). The nodes represent the input wires (x_1 , x_2 , y_1 , and y_2), registers (x , y , and z), and one output wire (out) with their node type, bit-widths, and connection information annotated as graph features. This information is preserved so that the original design can be reconstructed from circuitIR. This lossless conversion allows us to implement a

bi-directional transformation between HDLs and circuitIR, which is unique to previous works and is essential for SNS v2. With this feature, we can augment our dataset and use hierarchical graph neural networks with multi-step fine-tuning, as detailed in Sections 3.2 and 4.2.

3.2 Hierarchical GNNs

Previous works [29, 34, 35] have mainly relied on GNNs to predict synthesis results from the hardware design's graph intermediate representation. Although this approach can be both simple and effective, it still has its limitations. For instance, the GNN model operates in an "end-to-end" fashion, directly predicting the design's physical characteristics from its graph intermediate representation. The end-to-end GNN approach has a significant drawback: its inability to capture the hierarchical nature of RTL circuit designs. RTL designs have a hierarchical structure that provides valuable insights for extracting features from the designs. The entire circuit graph can be divided into two levels of hierarchy: combinational logic between nodes that are input/output wires and registers, and the data dependencies between these nodes and other similar nodes. Taking into account these hierarchical aspects can lead to a more comprehensive understanding of RTL designs, enabling improved analysis and optimization techniques.

The end-to-end GNNs also struggle to scale to large designs, and the model must either compromise at the level of abstraction or at the hardware design size. For instance, utilizing a higher abstraction to represent a hardware design results in a smaller graph intermediate representation, but the detailed features of the design are lost (e.g., D-SAGE [29] operates at an HLS semantic level).

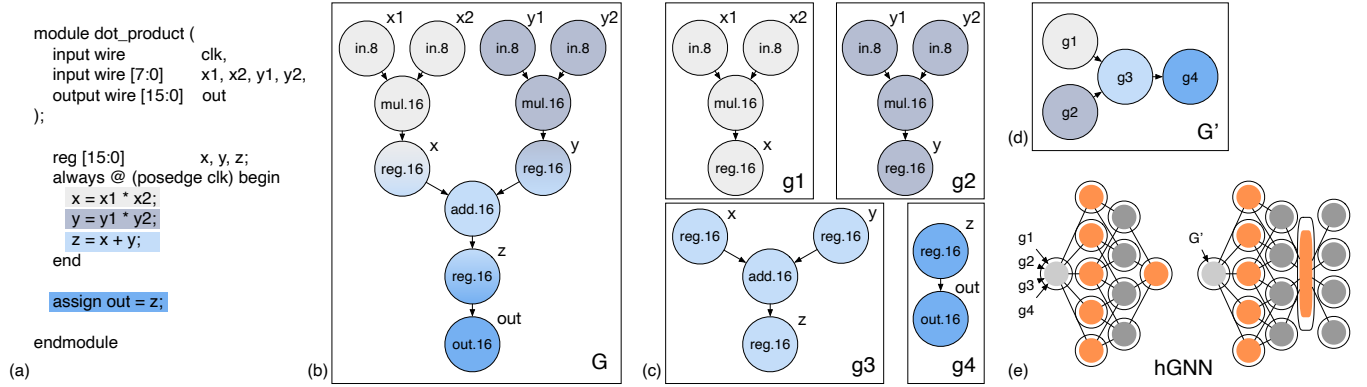


Figure 3: CircuitIR of a dot product circuit.

Prior work GRANNITE [40] can only predict power consumption for hardware design sizes up to a small RISC-V core. In practice, when predicting synthesis results, larger hardware designs are of greater interest as they generally take longer to synthesize.

Hierarchical Graph Neural Networks (HGNNs) have recently emerged as a powerful tool that leverages hierarchical graph representation learning to efficiently capture the hierarchical structure of graphs. By organizing nodes into hierarchical tiers and encoding clusters of nodes before examining the entire graph, HGNNs can simultaneously learn both local and global features within the graph, hence its recent emergence. This model is particularly relevant to the analysis of hardware designs, as it can effectively incorporate the inherent hierarchical nature of these RTL circuits. Furthermore, HGNNs demonstrate impressive scalability when applied to large graphs, as the size of intermediates at a single level of hierarchy is generally constrained.

3.3 Subgraph Sampling

In SNS v2, we leverage a two-level hierarchical structure to model RTL circuit designs, which consists of decomposed DAGs (Directed Acyclic Graphs) from large designs we call *subgraphs* and a dependency graph between the subgraphs. This approach is built on similar concepts as the original SNS [37], which samples circuit paths beginning and ending with register-type nodes or IO-type nodes. However, instead of sampling circuit paths, SNS v2 samples circuit subgraphs of the entire design with in, out, and reg nodes as roots and leaves, and purely combinational logic in between. By doing so, SNS v2 can incorporate domain-specific knowledge of RTL designs into the HGNN model, leading to improved analysis and optimization techniques.

Algorithm 1 illustrates how SNS v2 samples subgraphs from the initial circuit graph generated via the circuit graph compiler. The BFS algorithm starts from an out node or a reg node, and it backtracks through the graph until it reaches another in or reg node. The nodes traversed during this process form a subgraph. By using this method, we can sample a subgraph for every out or reg node in the entire circuit graph. While more complex graph partitioning and sampling methods exist, we have found that using a simple BFS back-tracing algorithm is sufficient for our purposes. This algorithm provides faster computations compared to other

Function BFSBackTrace(s):

```

V ← {s}; // Set of visited nodes
Q ← {s}; // Queue with start node
while Q ≠ ∅ do
  u ← dequeue Q; // Current node
  foreach v ∈ u.outgoing() do
    if type(v) ∈ {reg, in} then
      return v; // Found register-typed node
    end
  if v ∉ V then
    Q ← Q ∪ {v}; // Add unvisited node to queue
    V ← V ∪ {v}; // Mark node as visited
    v.setParent(u); // Set parent node
  end
end
return null; // No register-typed node found

```

Algorithm 1: Customized BFS with Backtracing

advanced algorithms as sampling from different register nodes is entirely independent, allowing ample parallelism and thus higher processing throughput.

This approach also enables the efficient capture of the hierarchical characteristics of RTL circuit designs. Consider the dot product graph representation shown in Figure 3(b). To obtain a set of subgraphs, the algorithm launches a BFS from each reg nodes, x, y, and z, and the only output node out. The algorithm then outputs four subgraphs, g1, g2, g3, and g4 respectively, as shown in Figure 3(c). These subgraphs can then be fed into the first level of the hierarchical GNN we call the *subgraphGNN*, which produces an embedding vector for each subgraph. However, to form a circuit-level prediction, we need to combine the embedding vectors of the subgraphs. This is where the second level of hierarchy in RTL circuit designs, the data dependencies between subgraphs, can be incorporated.

The BFS back-tracing procedure provides a way to construct a subgraph dependency graph (denoted as G' shown in Figure 3(d))

for the entire design, with nodes representing subgraphs and edges representing data dependencies between subgraphs. We integrate the subgraph embedding vectors, generated via the first-level subgraphGNN, by attaching embedding vectors as features to the nodes g_1 , g_2 , g_3 , and g_4 in the dependency graph G' . By augmenting G' with the embedding vectors we can use a second-level GNN we call the *circuitGNN* to produce more accurate circuit-level predictions as shown in Figure 3(e).

SNS v2 leverages the strengths of HGNNs and domain-specific knowledge of RTL circuits to provide an efficient and accurate method for predicting synthesis results. By incorporating data dependencies and taking advantage of the hierarchical nature of RTL circuits, SNS v2 outperforms traditional end-to-end GNN approaches, especially when applied to large-scale circuit designs.

3.4 Pre-training of the SubgraphGNN

Data scarcity presents a significant challenge in applying deep learning models to hardware design. High-quality open-source hardware designs are rare, and even when available, pre-processing, synthesizing, and implementing the designs using specific technology libraries require substantial computational resources and human effort.

To address this challenge, SNS v2 leverages abundant unlabeled data to enhance model performance through pre-training. Effective pre-training often depends on domain-specific hypotheses about input data, as seen in the groundbreaking Word2Vec technique [19] used in natural language processing. This technique is based on the assumption that words appearing in similar contexts tend to share similar meanings. This hypothesis, often called a domain-specific *distributional hypothesis*, allows Word2Vec to train a model that represents words as high-dimensional vectors that capture their semantic and grammatical relationships.

The primary innovation of SNS v2 lies in exploiting a domain-specific distributional hypothesis that functionally equivalent circuits can be represented with similar embedding vectors. EDA tools and hardware compilers offer insights for formulating a distributional hypothesis. An essential step in the design compilation process involves circuit optimization, which relies on a set of legal transformations that convert a circuit into an equivalent, lower-cost circuit. Two circuits are considered functionally equivalent if they exhibit identical functional behavior but have different intermediate representations (i.e., different circuitIR graph structures). We say that these two circuits are *functionally equivalent*.

If a model can generate similar representation vectors for two functionally equivalent circuits – those vectors being continuous, fixed-size representations that capture the essential features and properties of the circuits – the model can learn how compilers and EDA tools optimize circuits and identify equivalent circuits despite differences in their circuitIR structures. By contrast, if the model generates different representation vectors for two inequivalent circuits, it becomes more robust to noise and can focus more directly on circuit functionality, which results in improved performance and generalization. This observation can be formalized as the *distributional hypothesis for pre-training SNS v2: if two circuits are functionally equivalent, modern synthesis tools can optimize them into similar cost forms with high probability*.

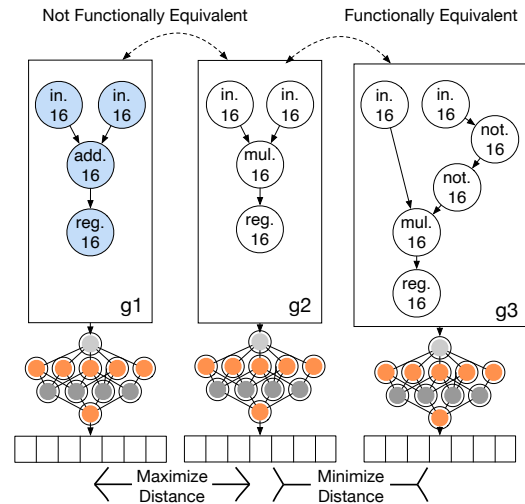


Figure 4: Contrastive pre-training for SNS v2: The model is trained to minimize the distance between the subgraph embeddings of functionally equivalent circuits while maximizing the distance between the subgraph embeddings of inequivalent circuits.

To pre-train SNS v2 using the formulated distributional hypothesis using unlabeled data, we employ a self-supervised learning technique called *contrastive learning*. Contrastive learning trains a neural network to learn input data representation by contrasting positive and negative examples. By maximizing the similarity between positive examples while minimizing the similarity between negative examples, contrastive learning compels the network to differentiate between two similar inputs from different classes. This technique has been widely adopted in both natural language processing and computer vision.

In the SNS v2 model, we apply contrastive pre-training to the first-level subgraphGNN model (described in Section 3.2). Rather than fitting the model with a known label, the training objective is to minimize the distance between the subgraph embeddings of functionally equivalent circuits and maximize the distance between the subgraph embeddings of inequivalent circuits. As illustrated in Figure 4, subgraphs g_1 and g_2 are not functionally equivalent because g_1 performs an add operation while g_2 performs a mul. operation. The model will maximize the distance between these two subgraph sample embeddings using contrastive learning. Conversely, subgraphs g_2 and g_3 do not have similar graph structures but are actually functionally equivalent, as the two not operations in g_3 cancel each other out, forming a circuit that is equivalent to g_2 . Hence, we seek to minimize the distance between their subgraph embeddings.

Using the contrastive pre-training technique in our model, we can achieve two critical benefits. First, we can learn a representation that is invariant to legal transformations, which can help to further improve the model’s robustness to noise and variability in the data. Second, since contrastive learning is a self-supervised

Table 1: Designs Collected

Category	Examples	Amount
GP. Core	RocketCore [2], BoomCore [42], XiangShan (Nanhu and Yanqihu) [30, 38]	20
SoC	4xRocket Soc	4
NoC	RingBus [41], MeshBus	16
Mem/IO	OpenHMC [25], AXI-4 Controller [1]	6
DSP	FFT [1], StreamingFIR [1]	8
Vector/Neural Accel.	Gemmini [7], VTA [20], Hwacha [24]	6
Crypto Core	AES, Sha3	3
Arithmetic	FPU (from hardfloat [2] and fudian [38]), SPMV [23], Gemm	16
Miscellaneous	LookupTable, BFS	10
Total:		89

technique, it enables us to train on unlabeled data, which is particularly important given the scarcity of labeled hardware design data.

3.5 Fine-tuning of the CircuitGNN

After the pre-training of the subgraphGNN model, the sampled subgraphs are encoded into embedding vectors. Once we have the embeddings, we construct a dependency graph that captures the data dependencies between the subgraphs via register dependencies. The dependency graph along with the embeddings are fed into the second-level circuitGNN model to predict hardware synthesis results. To ensure rapid inference speed, we employ simple graph convolutional network (GCN) layers [14] for hardware synthesis result prediction. For more information on the data collection method, neural network model implementation, and training process, please refer to Sections 4 and 5.

4 DATASET COLLECTION

In this section, we detail how we generate the datasets to pre-train and fine-tune our two-level HGNN model as well as what the datasets entail.

4.1 RTL Synthesis Dataset Generation

We start our data collection process by first collecting available open-source hardware designs and generating the “ground truths” by completing the synthesis tool flow using the industry standard Synopsys Design Compiler [26]. In addition to collecting open-source designs, we implement other popular designs and include them in the dataset. Table 1 lists the designs that are included. We use three technology libraries: Synopsys SAED-90nm, SAED-32nm, and SAED-14nm [8], and set various synthesis parameters (e.g., clock period, threshold voltage, temperature) and process corners. We create three synthesis configurations per library: *fast*, *typical*, and *slow*. The *fast* configuration enforces tight timing constraints, the lowest threshold voltage and temperature, and optimal process corners to obtain the fastest and most optimal result. Conversely, the *slow* configuration utilizes loose timing constraints, the highest threshold voltage and temperature, and the worst-case process corners. The *typical* configuration falls between the two. The generated synthesis results are used as the labels for our *circuit-level dataset* (Figure 5(a)). Note that the datasets generated using the 90nm and 14nm technology libraries are only used to assess SNS v2’s prediction accuracy, as the HGNN are pre-trained and fine-tuned using only the 32nm dataset.

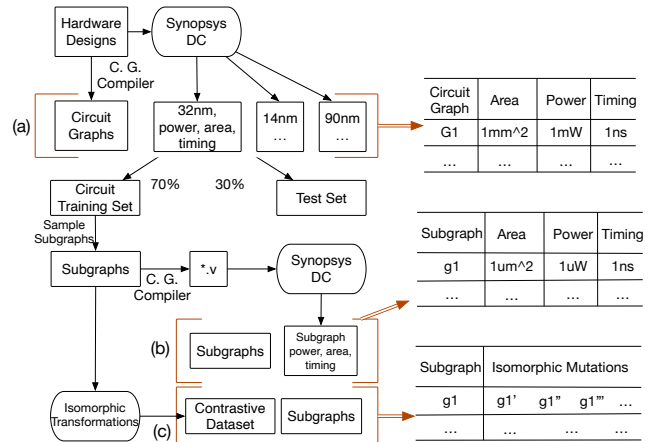


Figure 5: Dataset Generation Flow Chart. “C. G. Compiler” refers to the circuit graph compiler discussed in Section 3.1. Three datasets are generated for SNS v2: (a) the circuit-level dataset, (c) the subgraph pre-training dataset, and (b) the subgraph fine-tuning dataset.

To ensure fairness during testing, we divided our designs into training and test sets using a 7:3 ratio. We also imposed a restriction that any design in the test set could not be a submodule of any design in the training set and vice versa. For example, if the RocketCore design appears in a training set, the 4xRocketCore SoC design cannot be in the test set.

4.2 SubgraphGNN Dataset Generation

With all the designs collected, we generate the *pre-training dataset* and the *fine-tuning dataset* for the subgraphGNN by compiling the collected hardware designs into circuitIR and sample subgraphs from the designs. These subgraphs are all sampled from the training set of the circuit-level dataset so that the evaluation remains fair (i.e., no information leakage from the training set to the test set).

To enable contrastive learning, we need to create a contrastive dataset (i.e., the pre-training dataset) consisting of functionally equivalent circuitIR samples. As described in Section 3.4, we perform contrastive pre-training on the subgraph model, and therefore, we generate the contrastive dataset for the sampled subgraphs by creating “mutations” of each subgraph that are functionally equivalent. We identify a set of commonly used legal transformations in EDA tools and hardware compilers, randomly select a transformation from the list of legal transformations¹, and apply the transformation to the circuit. We set a parameter k such that a random transformation is selected and applied to the circuit k times. The larger the k , the more transformations are applied, and the more significant the mutation of the original circuit. We run this algorithm multiple times to generate a set of functionally equivalent circuits for each subgraph. In practice, we set $k = 3$ and

¹We implemented the following transformations: Common Subexpression Elimination, Constant Folding and Propagation, De-Morgan’s Law, Associative Law, Commutative Law, Dead Code Elimination, Mux Tree Re-ordering, Boolean Logic Optimization, Reduction Consolidation, Equivalent Comparison, and Shift to Multiplication.

Table 2: Model Architecture and Training Hyperparameters

Training Phase	First-Level subgraphGNN		Second-Level circuitGNN
	Pre-Training	Fine-Tuning	Domain-Adaptation
#GCN Layers	3		3
Hidden Dim	256		128
Batchsize	128	64	2
Optimizer	Adam [13]	Adam	Lion [4]
LR	1e-5	1e-4	5e-4
Dataset Size	24576	4096	58
#Epochs	10	100	400
Training Time	4 (h)	2.1 (h)	0.4 (h)

generated five new functionally equivalent circuits for each subgraph². In total, we sampled 4096 subgraphs and constructed 20480 functionally equivalent sets as our pre-training dataset for the subgraphGNN. This dataset is depicted in Figure 5(c) where subgraph g_1 has corresponding equivalent mutations g_1' , g_1'' , and so on.

After pre-training, the subgraphGNN is fine-tuned with labeled data, and therefore needs “ground-truths” by obtaining synthesis results via Synopsys DC. Though we’ve generated synthesis results for the circuit-level dataset, the results are for entire hardware designs and not for subgraphs. Recall that our circuit graph compiler has the capability to convert the sampled graph intermediate representation circuitIR back to FIRRTL, allowing the FIRRTL compiler to reconstruct HDL codes that are synthesizable. The subgraph HDLs are then synthesized using Synopsys DC to generate labeled synthesis data (i.e., timing, area, and power). This process of generating the fine-tuning dataset and the dataset itself are depicted in Figure 5(b) and the dataset sizes are listed in Table 2. Our results show that SNS v2 achieves high accuracy even with a small domain-adaptation dataset.

5 MODEL TRAINING AND EVALUATION

5.1 Methodology

To train our models, we implemented neural network architectures using the PyTorch library with GCN layers imported from the DGL Library [31]. The PyTorch Lightning framework coordinated the training methodology, while a single Nvidia A5000 GPU powered the training of all models. The selection of model hyperparameters, optimizers, as well as training times of the three phases are listed in Table 2.

5.2 Training of the subgraphGNN

Our first-level subgraphGNN is first pre-trained on an unlabeled equivalent subgraph dataset, the pre-training dataset, in a contrastive fashion to obtain vector embeddings that represent functionally equivalent circuits with similar vectors. The subgraphGNN is then fine-tuned using the labeled fine-tuning dataset with real synthesis results.

During the contrastive pre-training phase, we use a triplet training method where for each step, we perform the following: 1) randomly sample an anchor subgraph G_a , 2) randomly sample a positive sample G_p from G_a ’s equivalent set, and 3) randomly sample a negative sample G_n from a set that is not in the same equivalent set as G_a and therefore is not functionally equivalent to G_a . For

²We ran experiments using $k=2, 3, 4$, and 5 and empirically chose $k=3$ because it achieved the lowest contrastive training loss and facilitated a fast convergence of the first-level GNN model. The detailed training procedure is discussed in Section 5.

example, if we use subgraph g_1 as an anchor, then any mutation of g_1 can be a positive sample (e.g., g_1'), and any other subgraph that is not in the g_1 equivalent set can be a negative sample.

The objective is to minimize the triplet loss function [10]:

$\mathcal{L}_{triplet} = \max(0, \|\vec{E}_a - \vec{E}_{pos}\| - \|\vec{E}_a - \vec{E}_{neg}\| + m)$. In this process, we contrast the embedding of the anchor subgraph G_a with the embedding of the positive sample G_p and the embedding of the negative sample G_n . We want to minimize the distance between the anchor and positive sample representation in the latent space, while simultaneously maximizing the distance between the anchor and negative sample representation. This approach effectively encourages the learning of meaningful and discriminative features that distinguish between functionally inequivalent circuits.

Following the pre-training phase, we fine-tune the subgraphGNN using actual synthesis results from the EDA tools, specifically the Synopsys Design Compiler. This process begins with the checkpoint of the pre-trained model and the model is trained on the fine-tuning dataset with the object of minimizing the MAPE (Mean Absolute Percentage Error) loss function: $MAPE = \frac{100}{n} \sum_{t=1}^n \left| \frac{y - \hat{y}}{y} \right|$. After the fine-tuning phase, the first-level subgraphGNN is able to produce better embeddings of the circuit with the synthesis result knowledge baked in.

5.3 Training of the circuitGNN

SNS v2’s first-level subgraphGNN is capable of encoding any circuit subgraph into a representation vector that captures its structural, functional, and physical characteristics such as power, area, and timing. Using this vector, the second-level circuitGNN is trained on the circuit-level dataset which includes both the circuit subgraph representation vectors and the synthesis results of the hardware designs as the labels. We also use the minimization of the MAPE loss function as the training objective.

We fine-tune the circuitGNN exclusively on the 32nm typical target, and the results are presented in Figure 6-middle-left, labeled as ‘**’, and in Figure 7 where we compare the predicted results of SNS v2 vs. the ground-truth results. To show the effectiveness of the three-phase training process, we also train a baseline GNN model (similar to the one depicted in Figure 2(e)) that directly predicts the synthesis results from the hardware design circuit graph. The baseline model’s accuracy is plotted in Figure 6-middle-left, labeled as ‘Baseline’. We observe that SNS v2 achieves a much better accuracy than the baseline model, which demonstrates the effectiveness of the three-phase training process.

5.4 SNS v2 Domain Adaptation

To demonstrate the model’s adaptability and generalization capabilities across various technology nodes, we predict designs implemented on other technology nodes through a transfer learning approach. Once SNS v2 is fine-tuned on one target technology node and one synthesis configuration with labeled data (in our case, 32nm typical), it can be adapted to other technology nodes and operating conditions with just a few labeled data samples. In our experiments, we used exactly one design to generate the dataset for domain-adaptation of other technology nodes and operating conditions. For example, to predict a hardware design in 14nm fast, we will use the actual synthesis results of a RocketCore (and its

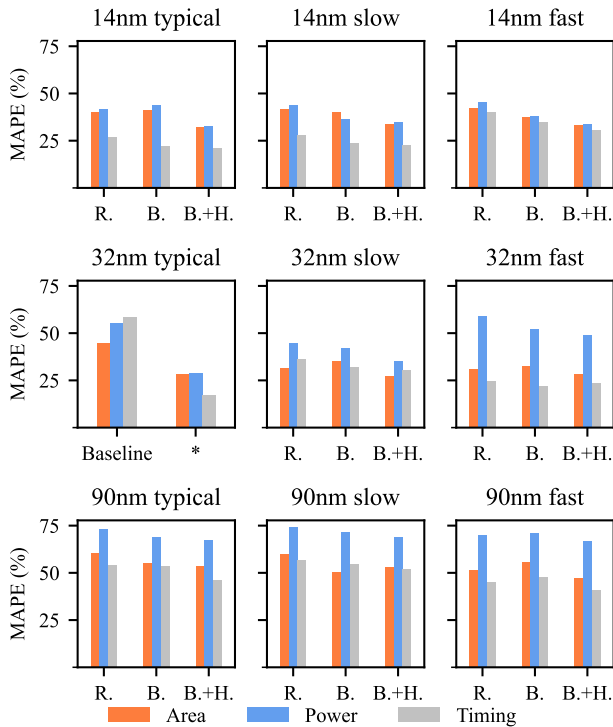


Figure 6: SNS v2 Synthesis Prediction Accuracy for both In-domain and Out-domain targets. 32nm typical is fine-tuned with the training set, which is labeled “*” in the figure. The baseline result is for an end-to-end model that directly predicts the synthesis result from the circuit graph. Other results are for the transfer learning approach, where the circuitGNN is fine-tuned on the 32nm typical target and then adapted to other technology libraries and synthesis corners with one data sample of a RocketCore (R.), a BoomCore (B.), or an SoC of BoomCore+Hwacha (B.+H.).

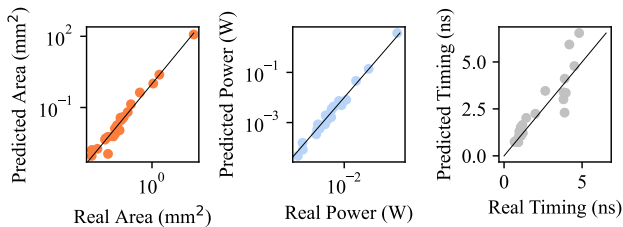


Figure 7: The scatter plot shows SNS v2’s predicted results vs. Synopsys Design Compiler’s ground-truth results for the 32nm typical target.

subgraphs) implemented using the 14nm technology library and a fast operating condition to perform domain-adaptation. The results are shown in Figure 6-top-right, labelled ‘R.’.

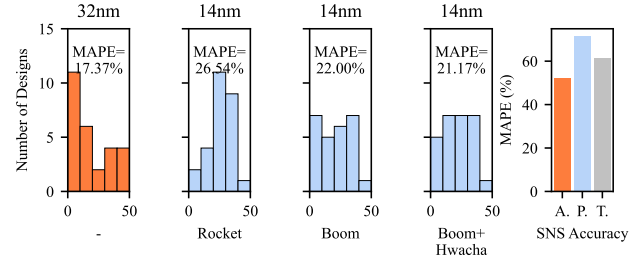


Figure 8: The distribution of the timing prediction’s Absolute Error Percentage (APE) when transferring from the 32nm typical to 14nm typical for three different domain-adaptation datasets: Rocket, Boom, and Boom+Hwacha. The error percentages are evaluated against the ground-truth synthesis result of each prediction target respectively (e.g., 32nm’s prediction result is evaluated against the 32nm’s ground-truth synthesis result while the 14nm’s prediction is evaluated against the 14nm’s synthesis result). Each bar represents the number of predictions that fall into the corresponding error percentage range (e.g., the leftmost bar in the leftmost figure for the 32nm’s prediction error indicates that there are 11 designs whose absolute prediction errors are within the range of 0-10%). The mean absolute percentage error (MAPE) for each distribution is annotated above the distribution bars. The figure on the right shows SNS [37]’s MAPE results (A: Area, P: Power, T: Timing). SNS v2 is able to achieve much better accuracy than SNS.

In our experiments, we consider three distinct domain adaptation datasets, each containing one design, covering a range of design sizes: a RocketCore (R.), a BOOMCore (B.), and a BOOMSoc with the Hwacha Vector Core (B.+H.). We performed domain adaptation across the 14nm, 32nm, and 90nm libraries, covering three unique synthesis corners – typical, slow, and fast, except for the 32nm typical corner, which served as our fine-tuning base.

Based on the outcomes, we can draw several conclusions: First, the performance of domain adaptation improves as the size of the fine-tuning dataset increases. While SNS v2 demonstrates satisfactory accuracy even with a small design like the RocketCore, it suggests that the model benefits from larger datasets during the domain-adaptation phase. This observation is further supported by Figure 8, which depicts the distribution of error percentages for the domain adaptation process in the timing prediction for the 14nm typical corner. Upon examining the error rates of using the smaller RocketCore design to the larger out-of-order BOOMCore, and an even larger BOOMSoc+Hwacha design, we noticed a substantial improvement in model accuracy (MAPE went from 26% to 21%), highlighting the model’s ability to capitalize on larger datasets for enhanced performance.

Second, we found that the 90nm library exhibits poorer domain-adaptation performance than the 14nm library. This may be attributed to significant differences between the cell modeling techniques used in the 90nm library differ greatly from those in the 14nm and

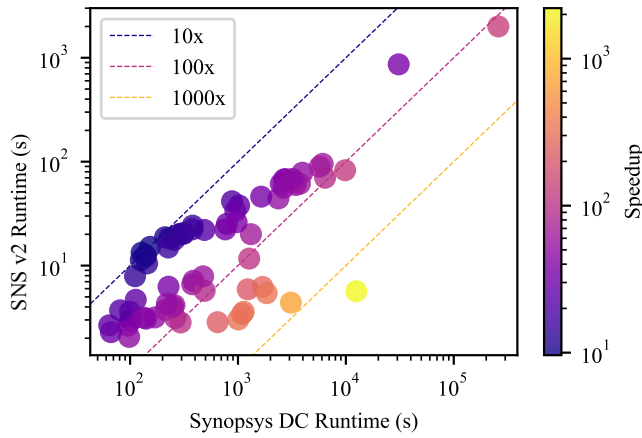


Figure 9: SNS v2 Speedup w.r.t. Synopsys Design Compiler

32nm libraries, the 90nm technology library uses NLDM (Non Linear Delay Model) model while both the 32nm and 14nm library relies on modern CCS (Composite Current Source) model. In reality, a MAPE as large as 67% means that the predicted result is expected to have an error of $\pm 67\%$ compared to the ground truth. At this level of accuracy, high-quality design space exploration results cannot be guaranteed by SNS v2. The user would need to further tune the SNS v2 model with a larger domain-adaptation dataset to improve accuracy.

Third, predicting power proves to be generally more challenging than predicting timing and area due to the accumulation of error in power and area predictions. This is because power is a function of both a circuit design’s timing and area.

5.5 Performance of SNS v2

SNS v2 achieves an average speedup of 411.2 \times over Synopsys Design Compiler. Figure 9 plots the designs we evaluated and the SNS v2 runtime when predicting the physical characteristics of the designs. Majority of the designs achieved at least two to three orders of magnitude speedup when predicted with SNS v2 over conventional Synopsys DC. It’s worth noting that the profiling for both SNS v2 and Synopsys DC was conducted on the same medium-sized server equipped with dual Xeon Gold 6252 CPUs and 512GB DDR4 Memory, without any GPU utilization for neural network inference, to ensure a fair comparison with Synopsys DC, which only runs on CPUs.

5.6 Comparison with SNS

Though the original SNS can achieve up to three orders of magnitude speedup compared to conventional EDA tools, SNS v2 achieves much better prediction accuracy than SNS (e.g., SNS v2’s timing MAPE of 17.37% vs. SNS’s MAPE of 61.46% as shown in Figure 8), provides transfer learning capability (i.e., using 32nm dataset to predict 14nm designs), and employs contrastive learning to improve its robustness using abundant unlabeled data. Although SNS provides a greater speedup on average (574 \times), SNS v2 provides significant advantage over SNS due to its better prediction accuracy, transferability, and robustness.

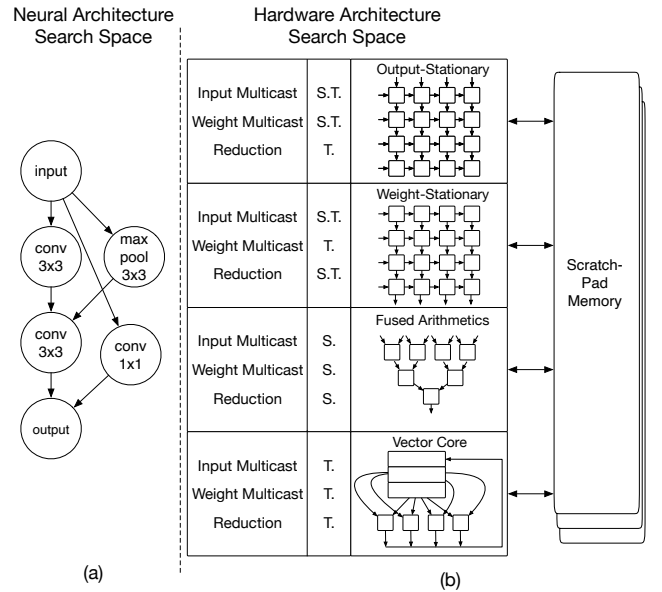


Figure 10: Neural and Hardware Architecture Search Space: (a) shows a sample from the neural architecture search space. (b) depicts the hardware architecture search space where we assign a type of neural accelerator core to each layer in the neural architecture and all the cores share a common scratchpad memory.

6 CASE STUDY

Neural Architecture Search (NAS) is a technique for automated designing of neural networks [6]. Optimizing neural network architectures for hardware implementation is crucial for resource-constrained environments such as edge devices. This is where hardware Neural Architecture Search (hNAS) comes in. By incorporating hardware constraints, hNAS generates efficient and effective neural network architectures for an optimal hardware implementation.

In this case study, we demonstrate the integration of SNS v2 into a neural accelerator generator to provide real-time feedback for improved hardware generation quality. We perform hNAS, where we search for the optimal neural network architecture and hardware configuration for an image classification task based on cost.

6.1 Neural Architecture Search Space

To search for the optimal neural network architecture, we utilize the NASBench101 dataset [39] as our search space. This dataset is comprised of 423,000 unique neural networks with varying architectures and hyperparameters, with each trained and evaluated on the CIFAR-10 dataset [16], recording both test and training accuracies.

The search space defined in NASBench101 is concentrated on small feedforward structures called *cells*, represented as DAGs with a maximum of 7 vertices and 3 possible operations: 3 \times 3 convolution, 1 \times 1 convolution, and 3 \times 3 max-pool. Constraints within the search space limit the number of vertices to 7, the number of edges to 9; only batch normalization with ReLU activation is supported. This search space enables the exploration of ResNet-like [11] and

Inception-like [27] cells while maintaining tractability. Figure 10(a) depicts an example of a neural network architecture cell in the NAS-Bench101 dataset. For this case study, we will be generating these cells via NAS and evaluating different hardware implementations of these cells via hNAS.

6.2 Hardware Architecture Search Space

We define our hardware architecture search space as a cluster of heterogeneous neural accelerator cores connected to a shared scratchpad memory as shown in Figure 10(b). For each layer in the neural network, we assign a dedicated neural accelerator core. We restrict each core to perform computations solely for its corresponding neural layer. To simplify our experiment, we make the assumption that the shared scratchpad memory is large enough to hold all required intermediate results, and all communications between neural accelerator cores occur through the scratchpad memory. The modeling of the scratchpad memory itself is beyond the scope of the SNS v2 model and will not be included in this case study.

The search space for one core in the cluster is defined through a simplified dataflow taxonomy proposed by the MAESTRO framework [17]. MAESTRO classifies neural network accelerators based on how the input and weight to each neural layer is multicasted and how the multiplication results are reduced to a single value. These implementations are categorized into three different dataflows: temporal, spatial, and spatial-temporal. A temporal multicast requires a data element to be latched in a register, which can be accessed over multiple cycles. A spatial multicast broadcasts data to all processing elements (PEs) in a single cycle with a direct fan-out. The spatial-temporal dataflow combines both spatial and temporal dataflows by allowing one piece of data to hop from one PE to another each cycle.

To produce the most common types of neural accelerator cores based on this taxonomy, we implemented four types of accelerator cores: vector machine, fused arithmetic, weight stationary, and output stationary as shown in Figure 10(b). We developed a vector core, a fused core, and a systolic array generator in Chisel [3], supporting these four types of neural accelerator cores with the number of PEs being 16, 64, 256, or 1024. Systolic arrays are constructed in square shapes (e.g., 16 PEs corresponds to a 4×4 systolic array) while the number of PEs corresponds to the number of multipliers for the vector core and the fused core. Our performance model in Python accurately produces cycle counts for each type of neural accelerator core with a specific number of PEs for a given neural network layer.

6.3 Search Results and Evaluation

To conduct a hardware neural architecture search, we must define a specific optimization objective for the search problem. In our case study, we aim to minimize the *cost per inference* regularized with the neural network’s accuracy. The regularization is necessary as we want to reward the search algorithm to find a network that’s both low-cost and high-accuracy³. We aim to search for both the

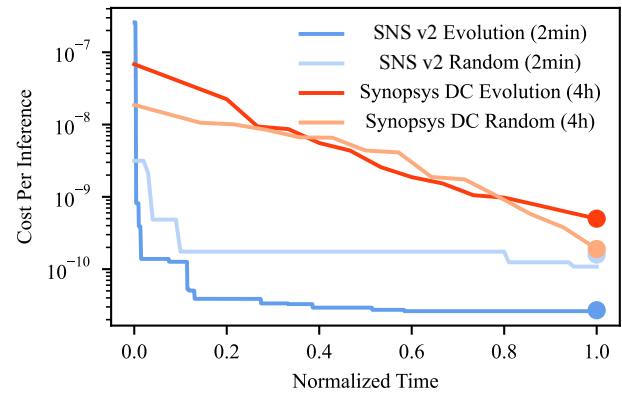


Figure 11: SNS v2 vs. Synopsys Design Compiler for Providing Guidance for Hardware Generator in Hardware Neural Architecture Search. While SNS v2 predicts the synthesis result, there may be some errors in the prediction. To ensure a fair comparison, the search result of the SNS v2 model is evaluated by synthesizing the hardware architecture generated using the Synopsys Design Compiler at the end. The final evaluation result is depicted as dots on the right side of the figure.

optimal neural architecture and the optimal hardware architecture that supports it.

We implement two search algorithms for our case study: Random Search and Evolutionary Search. The Random Search algorithm first randomly samples a neural architecture from the search space, and is followed by randomly sampling a hardware architecture for that neural architecture. The Evolutionary Search algorithm starts with a population of randomly sampled neural architectures and hardware architectures, and then iteratively updates the population by applying mutations over the population. The mutations involve changing both the neural architecture and the hardware architecture. The neural architecture mutations involve adding or removing edges or nodes from the computation graph, effectively transforming the neural network topology. In contrast, hardware architecture mutations involve adjusting the number of PEs and the dataflow type. By altering these parameters, the algorithm strives to optimize the hardware’s efficiency and throughput, catering to the specific demands of the neural architecture and the target application.

We perform the two types of searches for fixed wall-clock time periods using both SNS v2 and Synopsys DC and compare the cost per inference as the search converges shown in Figure 11. We observe that the Evolutionary Search algorithm not only converges much faster than the Random Search algorithm, but also produces better results. Furthermore, our SNS v2 model outperforms Synopsys DC in both speed and quality. In just 2 minutes, SNS v2 finds a solution that is an order of magnitude better than the Synopsys DC solution, which takes 4 hours to discover. This significant improvement in speed and quality has substantial benefits for complex large-scale hardware generators, as it enables real-time design decisions based on the hardware cost feedback provided by SNS v2.

³We make the following assumptions: (1) Average electricity price: 0.2/kWh. (2) Chip fabrication costs: 0.05/mm² for 32nm. (3) We regularize the cost based on the network’s accuracy through a regularization factor $\sigma = (0.9)^{(\text{accuracy}-0.9)/0.01}$. (4) A chip lasts 5 years before obsolescence and operates under full load for 40% of that time.

	Model Input	Prediction Target	Transferability	Scalability	Arbitrary Design
Aladdin [15]	Aladdin DSL	Power, Performance	Not ML based	No limit	No
MAESTRO [17]	NPU Specs	Power, Performance	No	No limit	No
GRANNITE [40]	Netlist w. Traces	Dynamic Power	No	mini-RISC-V Core	Yes
ECO-GNN [18]	Netlist w. Traces	Dynamic Power	Across workloads	RocketCore	Yes
D-SAGE [29]	HLS	Timing, Routability	No	Micro-kernels	No
SNS [37]	RTL	Power, Area, Timing	No	RISC-V SoC	Yes
SNS v2	RTL	Power, Area, Timing	Across TechLibs	RISC-V SoC	Yes

Table 3: Comparing SNS v2 with related work. The scalability column shows the largest design size in the model’s test set.

7 RELATED WORKS

In this section, we provide a comparison of our work and other related works. We summarize them qualitatively in Table 3.

7.1 Pre-RTL Hardware Cost Estimators

Pre-RTL hardware cost estimators like Aladdin [15] allow for quick hardware cost estimation without having the RTL code ready. However, these estimators suffer from limited scope and accuracy due to lost of implementation details. To maintain accuracy, pre-RTL models impose constraints on input domains. The scope of these estimators is also limited to specific domains, such as neural network accelerators. The Aladdin framework is limited to accelerators conforming to the SoC system defined by the framework. In contrast, SNS v2 can predict hardware cost for any arbitrary RTL hardware designs.

7.2 GNN Predictors for Hardware Designs

GNNs are used to predict hardware design characteristics, such as timing delays and power consumption. D-SAGE is a GNN-based model that predicts timing delays and routability on FPGAs using graph structures learned from HLS code. GRANNITE [40] and ECO-GNN [18] are GNN-based models that predict power consumption based on graph representations, with GRANNITE demonstrating speed advantages but scaling poorly with input size, while ECO-GNN uses a subgraph approximation technique to reduce training and inference costs. SNS v2 adopts a similar GNN-based model inspired by D-SAGE but uses a hierarchical approach to training and inference for large designs in order to provide scalability.

8 CONCLUSION

In this paper, we presented SNS v2, a fast, robust, and transferable hardware synthesis predictor based on deep learning models. By utilizing equivalent pre-training and a three-phase training approach, SNS v2 achieves state-of-the-art accuracy and transferability compared to other works. Furthermore, we demonstrate the applicability of SNS v2 within a neural accelerator generator framework. In this context, SNS v2 provides real-time guidance for design space exploration, resulting in both faster speeds and improved quality of results compared to conventional EDA tools.

ACKNOWLEDGMENTS

We thank the reviewers for their insightful comments and suggestions. This work was supported in part by an National Science Foundation CAREER award CCF-2045973, in part by an National

Science Foundation AI Institute Athena AI-Driven Next-generation Networks at the Edge award CNS-2112562-002 & 004, and in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA.

REFERENCES

- [1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. <https://doi.org/10.1109/MM.2020.2996616>
- [2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).
- [3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference (San Francisco, California) (DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. <https://doi.org/10.1145/2228360.2228584>
- [4] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. 2023. Symbolic Discovery of Optimization Algorithms. <https://arxiv.org/abs/2302.06675>
- [5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 Long and Short Papers*. Association for Computational Linguistics, Minneapolis, Minnesota, USA, 4171–4186. <https://doi.org/10.18653/v1/N19-1423>
- [6] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *The Journal of Machine Learning Research* 20, 1 (2019), 1997–2017.
- [7] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. <https://doi.org/10.1109/DAC18074.2021.9586216>
- [8] R. Goldman, K. Bartleson, T. Wood, K. Kranen, C. Cao, V. Melikyan, and G. Markosyan. 2009. Synopsys’ open educational design kit: Capabilities, deployment and future. In *2009 IEEE International Conference on Microelectronic Systems Education*. 20–24. <https://doi.org/10.1109/MSE.2009.5270840>
- [9] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, San Francisco, California, USA, 4171–4186. <https://doi.org/10.1145/2939672.2939754>
- [10] Mai Lan Ha and Volker Blanz. 2021. Deep ranking with adaptive margin triplet loss. *arXiv preprint arXiv:2107.06187* (2021).
- [11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA, 770–778. <https://doi.org/10.1109/CVPR.2016.90>

- [12] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. <https://doi.org/10.1109/ICCAD.2017.8203780>
- [13] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]
- [14] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR abs/1609.02907* (2016). arXiv:1609.02907 <http://arxiv.org/abs/1609.02907>
- [15] Yunyong Ko, Kibong Choi, Hyunseung Jei, Dongwon Lee, and Sang-Wook Kim. 2021. ALADDIN: Asymmetric Centralized Training for Distributed Deep Learning. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. Association for Computing Machinery, Virtual Event Queensland Australia, 863–872. <https://doi.org/10.1145/3459637.3482412>
- [16] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. [n. d.]. CIFAR-10 (Canadian Institute for Advanced Research). ([n. d.]). <http://www.cs.toronto.edu/~kriz/cifar.html>
- [17] Hyoukjun Kwon, Prasanth Charatari, Michael Pellauer, Anshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) (*MICRO '52*). Association for Computing Machinery, New York, NY, USA, 754–768. <https://doi.org/10.1145/3352460.3358252>
- [18] Yi-Chen Lu, Siddhartha Nath, Sai Pentapati, and Sung Kyu Lim. 2022. ECO-GNN: Signoff Power Prediction Using Graph Neural Networks with Subgraph Approximation. *ACM Trans. Des. Autom. Electron. Syst.* (oct 2022). <https://doi.org/10.1145/3569942> Just Accepted.
- [19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) (*NIPS'13*). Curran Associates Inc., Red Hook, NY, USA, 3111–3119.
- [20] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16. <https://doi.org/10.1109/MM.2019.2928962>
- [21] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] <https://arxiv.org/abs/2303.08774>
- [22] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. 2022. Hierarchical Text-Conditional Image Generation with CLIP Latents. arXiv:2204.06125 [cs.CV] <https://arxiv.org/abs/2210.10606>
- [23] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 110–119. <https://doi.org/10.1109/IISWC.2014.6983050>
- [24] Colin Schmidt, Alber Ou, and Krste Asanović. 2018. Hwacha v4: Decoupled data parallel custom extension. In *Proc. Inaugural RISC-V Summit*. 1–40.
- [25] Juri Schmidt and Ulrich Bruning. 2015. openHMC—a configurable open-source hybrid memory cube controller. In *2015 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 1–6.
- [26] Inc. Synopsys. [n. d.]. DC Ultra: Concurrent Timing, Area, Power and Test Optimization. Synopsys, Inc. <https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html>
- [27] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA, 1–9. <https://doi.org/10.1109/CVPR.2015.7298594>
- [28] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] <https://arxiv.org/abs/2302.13971>
- [29] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. 2020. Accurate Operation Delay Prediction for FPGA HLS Using Graph Neural Networks. In *Proceedings of the 39th International Conference on Computer-Aided Design (Virtual Event, USA) (ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 87, 9 pages. <https://doi.org/10.1145/3400302.3415657>
- [30] Kaifan Wang, Yanan Xu, Zihao Yu, Dan Tang, Guokai Chen, Xi Chen, Lingrui Gou, Xuan Hu, Yue Jin, Qianruo Li, Xin Li, Jiawei Lin, Tong Liu, Zhigang Liu, Huaqiang Wang, Huizhe Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Ziyue Zhang, Yangyang Zhao, Yaoyang Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Wei He, Ninghui Sun, and Yungang Bao. 2023. XiangShan Open-Source High Performance RISC-V Processor Design and Implementation. *Journal of Computer Research and Development* 60, 3 (2023), 476–493. <https://doi.org/10.7544/jssn1000-1239.202221036>
- [31] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2020. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. arXiv:1909.01315 [cs.LG]
- [32] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys—A Free Verilog Synthesis Suite.
- [33] Nan Wu, Hing Yang, Yuan Xie, Pan Li, and Cong Hao. 2022. High-Level Synthesis Performance Prediction Using GNNs: Benchmarking, Modeling, and Advancing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) (*DAC '22*). Association for Computing Machinery, New York, NY, USA, 49–54. <https://doi.org/10.1145/3489517.3530408>
- [34] Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. 2018. RouteNet: Routability prediction for Mixed-Size Designs Using Convolutional Neural Network. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. <https://doi.org/10.1145/3240765.3240843>
- [35] Zhiyao Xie, Rongjian Liang, Xiaoqing Xu, Jiang Hu, Chen-Chia Chang, Jingyu Pan, and Yiran Chen. 2022. Preplacement Net Length and Timing Estimation by Customized Graph Neural Network. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4667–4680. <https://doi.org/10.1109/TCAD.2022.3149977>
- [36] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Herbert, Jiang Hu, Yiran Chen, and Shidhartha Das. 2021. APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, New York, NY, USA, 1–14. <https://doi.org/10.1145/3466752.3480064>
- [37] Ceyu Xu, Chris Kjellqvist, and Lisa Wu Wills. 2022. SNS’s Not a Synthesizer: A Deep-Learning-Based Synthesis Predictor. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) (*ISCA '22*). Association for Computing Machinery, New York, NY, USA, 847–859. <https://doi.org/10.1145/3470496.3527444>
- [38] Yanan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, ZuoJun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Qian, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199. <https://doi.org/10.1109/MICRO56248.2022.00080>
- [39] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. 2019. NAS-Bench-101: Towards Reproducible Neural Architecture Search. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 7105–7114. <https://proceedings.mlr.press/v97/ying19a.html>
- [40] Yanqing Zhang, Haoxing Ren, and Brucec Khalany. 2020. GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. <https://doi.org/10.1109/DAC18072.2020.9218643>
- [41] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. 2022. Constellation: An Open-Source SoC-Capable NoC Generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. 1–7. <https://doi.org/10.1109/NoCArc57472.2022.9911299>
- [42] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, Vol. 5.