SNS's not a Synthesizer: A Deep-Learning-Based Synthesis Predictor

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

Chris Kjellqvist
christopher.kjellqvist@duke.edu
Duke University
Durham, North Carolina, USA

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

ABSTRACT

The number of transistors that can fit on one monolithic chip has reached billions to tens of billions in this decade thanks to Moore's Law. With the advancement of every technology generation, the transistor counts per chip grow at a pace that brings about exponential increase in design time, including the synthesis process used to perform design space explorations. Such a long delay in obtaining synthesis results hinders an efficient chip development process, significantly impacting time-to-market. In addition, these large-scale integrated circuits tend to have larger and higher-dimension design spaces to explore, making it prohibitively expensive to obtain physical characteristics of all possible designs using traditional synthesis tools.

In this work, we propose a deep-learning-based synthesis predictor called SNS (SNS's not a Synthesizer), that predicts the area, power, and timing physical characteristics of a broad range of designs at two to three orders of magnitude faster than the Synopsys Design Compiler while providing on average a 0.4998 RRSE (root relative square error). We further evaluate SNS via two representative case studies, a general-purpose out-of-order CPU case study using RISC-V Boom open-source design and an accelerator case study using an in-house Chisel implementation of DianNao, to demonstrate the capabilities and validity of SNS.

CCS CONCEPTS

• **Hardware → Integrated circuits**; **High-level and register-transfer level synthesis**; • **Computing methodologies → Neural networks**.

KEYWORDS

Integrated Circuits, RTL-level Synthesis, Neural Networks

ACM Reference Format:
Ceyu Xu, Chris Kjellqvist, and Lisa Wu Wills. 2022. SNS's not a Synthesizer: A Deep-Learning-Based Synthesis Predictor. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527444

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527444

1 INTRODUCTION

Accelerators face friction when being adopted by industry in a large scale. With emerging applications introducing different types and demands for computations over time, accelerators are sought after to meet these new computation demands. Unlike general purpose processors that are programmable, accelerators sacrifice generality for specificity to offer unmatched performance and efficiency. General-purpose processors can still accommodate the shift of computation demands as programs old and new can run without modifications. On the other hand, even though accelerators can provide two to three orders better efficiency processing emerging workloads, they are time-consuming to develop and difficult to program. The only way accelerators can keep up with the ever-changing computation demands is by having a fast development process and a relatively short design iteration cycle.

Specialized hardware are difficult to develop and optimize. Unlike software which can be modified, compiled, and tested quickly and easily, one major challenge that forbids the expedition and the simplification of accelerator development is its long feedback delay while waiting for synthesis results. In our experience, even synthesizing a 32×32 systolic array that supports bfloat16 input datatype and float32 accumulation with an area around 2.5mm^2 takes up to one day using the traditional synthesis toolchain such as the Synopsys Design Compiler [14]. With modern computer chips having areas (and design complexities) reaching hundreds or even thousands of mm^2 (e.g., NVIDIA A100 GPU at 826mm^2 [31], AMD Epyc Rome at 1000mm^2 [29]), the problem of long design iteration cycles exacerbates.

To solve this problem, the computer chip industry came up with two solutions: adopting a modular hardware design methodology and/or employing experienced hardware developers. By designing hardware in a modular fashion, each module can be smaller and takes less time to synthesize. The delay of knowing the synthesis results, though still painfully long, becomes more tolerable. Experienced hardware developers usually have enough hardware design knowledge to make a good guesstimate of the physical characteristics (e.g., cycle time) of the hardware blocks being developed. They do not need to solely rely on synthesis feedback at every step throughout the development process.

However, neither of these solutions solve the long synthesis runtime problem. In addition, they lead to other shortcomings in the design process. Utilizing the modular design methodology introduces the possibility that the developers lack a holistic view of the entire design and may design modules that are locally optimal but not globally optimal, compromising the quality of the design. Human experiences are not always reliable, especially when new

fabrication technologies emerge. With these new technologies, different components may exhibit very different scaling behaviors, and hardware design experiences might even be harmful when the guesstimates are wildly incorrect under certain circumstances.

In this work, we present a deep-learning-based synthesis predictor called SNS that not only makes more accurate area, power, and timing predictions than prior work and also runs at two to three orders of magnitude faster than traditional synthesis tools.

Leveraging recent advances in artificial intelligence and graph analytics, SNS *turns input designs into graph representations* for synthesis predictions. Inspired by the path-based approach for social network analysis [24], SNS takes a novel *circuit-path-based approach* to predict the physical characteristics of individual circuit paths and aggregating the paths' characteristics to predict the area, power, and timing of the entire input design. Leveraging sequence processing techniques that learns the order and the placement of words in relation to a sentence such as natural language processing [36], SNS *learns the order and the placement of functional units* in a circuit path to provide more accurate synthesis result predictions. With very limited number of open-source hardware designs available as training data, SNS *utilizes generative models [41] to generate training datasets*, providing accurate predictions even when training data is scarce.

This paper makes the following contributions:

- An ultrafast and accurate (more accurate than D-SAGE [32]) synthesis predictor called SNS. SNS predicts the area, power, and timing of arbitrary input designs in seconds per design (on average 760× faster than the Synopsys Design Compiler for the designs we experimented) for design sizes and complexities ranging from a 128-entry 8-bit lookup table to a 16-core accelerator that computes single precision floating point 2D stencils.
- An augmented, light-weight Transformer model called *Circuitformer* that accurately predicts the physical characteristics of circuit paths.
- A 4000+ circuit paths training dataset for the Circuitformer that is generated using a Depth-First-Search-based random sampling algorithm along with a Markov Chain and a Generative Adversarial Network for sequences [41].
- Two case studies to demonstrate the capability, scalability, and validity of SNS: 1) a design space exploration based on a general-purpose out-of-order RISC-V processor core BOOM [43] to select three designs out of 2500+ designs on the Pareto frontier, and 2) a synthesis results comparison, a design space exploration, and a tradeoff study between datatype and model accuracy based on a classic machine learning accelerator DianNao [6].

2 BACKGROUND AND MOTIVATION

Machine learning (ML) models have shown their capability in fields such as image classification and natural language processing. Now computer architects too are beginning to consider how they might use ML to design chips. A variety of ML models have been proposed to predict different results of the chip design process. D-SAGE [32] proposed a custom GraphSage model for predicting the operation delay in an High-Level Synthesis (HLS) design. GRANNITE [42] uses a Graph Convolutional Network (GCN) model to predict the runtime power of a circuit design. PowerNet [38] proposed a Convolutional Neural Network (CNN) model for predicting the runtime voltage drop of a design. More traditional models have also been used. For example, Pyramid [22] used an ensemble of traditional ML models including Random Forest, Support Vector Machine (SVM), and Linear Regression to predict the resource usage of an HLS design on an FPGA.

Among these models, the Graph Neural Networks (GNN) models (GraphSage and GCN) stand out from other models because of their accuracy and elegance when performing circuit analysis. Hardware circuits are essentially graphs where the circuit's functional modules correspond to vertices and the wiring connections between them correspond to edges, making GNNs an obvious choice for performing circuit analysis.

But while they are elegant, they scale poorly for inference in large circuit graphs. Each layer of a GNN creates a graph embedding that incorporates the state of a node and its direct neighbors. In practice, circuit paths may be hundreds of nodes deep so a GNN capable of performing inference on full paths will also be hundreds of layers deep. Computing the gradient of a GNN with K layers requires storing the entire Kth-order neighborhood in memory, making the training process require an unreasonable amount of resouces. Further, while GNNs successfully perform inference for global graph properties, they are poor at inferring local properties of a graph; both of which are needed for circuit inference. For example, although being only a small portion of the design, the critical path determines the timing for the design and therefore affects the entire design's power. As a result, GNNs are impractical for performing inference on large-scale circuits.

2.1 Objective of SNS

As its name suggests, SNS is not a synthesizer and is neither meant to produce the most accurate synthesis result nor provide gate-level synthesized net-lists. SNS aims to reduce the delay of obtaining synthesis results and make high dimensional design space explorations feasible. These goals can be achieved with SNS's fast inference but are impossible if we cannot resolve the underlying model's reliance on large datasets for training. Therefore, we explore possible ways of training SNS with extremely scarce data.

2.2 Novelty of SNS

SNS attempts to solve the challenges in existing prediction models by taking an entirely different approach: SNS works on circuit paths rather than working on the graph representation of the entire design. Working on paths provides SNS with many advantages.

First, paths are faster and easier to process compared to graphs. The maximum length of a circuit path is around 500, which takes milliseconds to infer the properties of. This feature not only makes SNS ultrafast (on average, 760× speedup over Synopsys DC), but it also enables SNS to work on extremely large designs. Our results show that SNS scales perfectly to designs up to the size of 18 million gates (approximately 67.8 million transistors) [1]. Second, working

[1]The gate count and transistor count numbers are obtained from Yosys gate-level synthesis.

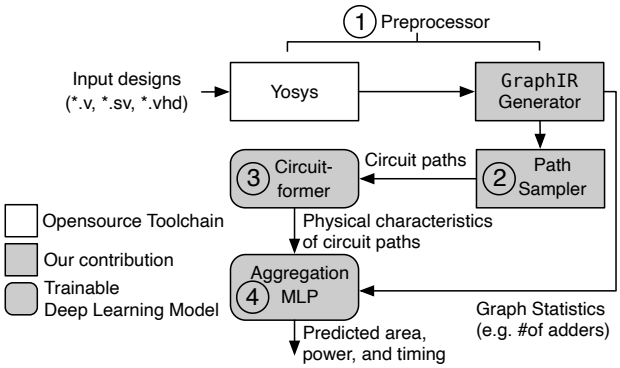


Figure 1: SNS Prediction Flow

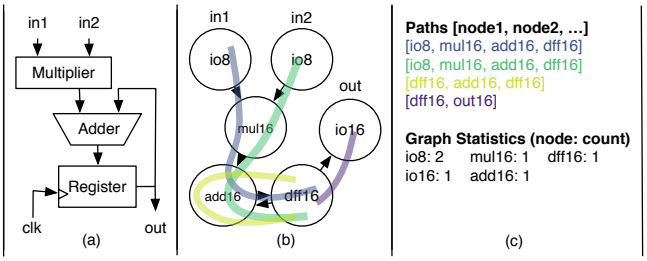


Figure 2: Transformation from Source Circuit to `GraphIR` Representation with Path Information and Graph Statistics

on paths allows SNS to infer "local properties" in the design, such as the critical path.

Since SNS only works on one path at a time, it can make accurate predictions for each path individually without interference from adjacent, unrelated modules. Third, working on individual paths enables SNS to trivially locate the critical path in the design. Knowing both the length and location of the critical path are important for improving the design. However, because GNNs perform inference globally, the individual properties of paths are obfuscated, making it hard to decide what path is chiefly responsible for an inference result. For SNS, since each path is explicitly sampled from the design, a record is kept for where each path is located in the design. Lastly, working on paths gives SNS the ability to use generative models to augment the dataset, which is a very helpful feature when the hardware design data is scarce.

3 THE DESIGN OF SNS: PATH-BASED SYNTHESIS PREDICTOR USING NEURAL NETWORKS

Modern machine learning models have found much success by inspecting features of an input in context. For instance, to perform imagine classification, Convolutional Neural Networks (CNNs) use convolutional filters to extract high-level features from a grid of pixels rather than looking at individual pixels. To analyze a social network, Graph Convolutional Networks (GCNs) learn features of all followers of an individual rather than focusing on the individual alone. To perform natural language processing, Transformers [33] use attention heads to extract a word's relationship to other words in the sentence, rather than trying to understand each word independently. Inspired by these popular and successful deep learning models, we propose SNS to predict physical characteristics of a design by learning attributes (e.g., area, power, and timing) not just from individual functional units (or individual nodes in a circuit graph) but complete circuit paths that consist of all nodes along those circuit paths. This allows SNS to first predict the physical characteristics of one circuit path in a design and then gradually aggregate the characteristics of multiple paths into design-level physical characteristics, eventually predicting the area, power, and timing of an input design in its entirety.

Figure 1 depicts an overview of the SNS prediction flow. A typical flow consists of four steps: ① compile and convert an input design into a graph intermediate representation (`GraphIR`) to represent circuits using the `GraphIR` *Generator* in the *Preprocessor*, ② sample complete circuit paths from the generated `GraphIR` using the *Path Sampler*, ③ use a trained Transformer model we call *Circuitformer* to predict physical characteristics of sampled paths, and ④ aggregate the path-level physical characteristics using a multi-layer perceptron neural network, *Aggregation MLP*, to predict area, power, and timing of the input design. For the remaining of this section, we describe in detail our design of SNS.

3.1 Turning Input Designs into Circuit Graphs

SNS takes input designs as circuits represented in the format of commonly used Hardware Description Language (HDL) source codes (i.e., system Verilog, Verilog HDL or VHDL). The first step of the SNS prediction is to compile the input design source codes and generate a graph intermediate representation of the circuit for further analysis. This graph intermediate representation is referred to as `GraphIR` or circuit graph for the remaining of the paper. The compilation is done using Yosys [34], an open-source synthesis suite that contains a complete tool chain from compiling HDL source codes to performing gate-level synthesis. SNS uses Yosys only for parsing and compiling the input design source codes into its circuit representation which includes all basic functional blocks (e.g., multiplexers, adders, multipliers) and their wiring connections.

The circuit representation is then processed using a `GraphIR` generator in the following manner. All input and output ports (e.g., `io`) as well all functional blocks and units (e.g., `mul`, `add`) are constructed as vertices (or nodes) in a circuit graph $G = (V, E)$; let's denote the vertices as $v \in V$. All connections between vertices are constructed as directed edges in the graph. Besides using the *type* of the nodes (e.g., `io`, `mul`, `add`) to differentiate vertices from one another, SNS also utilizes the bit-widths of the wiring connections (denoted as *width*) as another vertex property. Each node in the graph is named using a concatenation of the *type* and the *width*. For example, a 16-bit multiplier is represented as a vertex with the name `mul16`. Figure 2 shows pictorially how an 8-bit multiply-add unit from (a) is turned into a circuit graph in (b), having two 8-bit input ports `io8`, a 16-bit multiplier `mul16`, a 16-bit adder `add16`, a 16-bit register `dff16` (`dff` stands for D-flip-flop), and a 16-bit output port `io16`. The `GraphIR` generator also produces circuit graph statistics (shown in (c)) such as the counts of each distinct node names. The

Table 1: `GraphIR` Vertex Embeddings

Vertex/Functional Unit	*type*	*width*
Input/Output port	`io`	4, 8, 16, 32, 64
D-Flip-Flop	`dff`	4, 8, 16, 32, 64
Multiplexer	`mux`	4, 8, 16, 32, 64
Bitwise NOT	`not`	4, 8, 16, 32, 64
Bitwise AND	`and`	4, 8, 16, 32, 64
Bitwise OR	`or`	4, 8, 16, 32, 64
Bitwise XOR	`xor`	4, 8, 16, 32, 64
Parametrizable Shifter	`sh`	4, 8, 16, 32, 64
Reduced AND	`reduce_and`	4, 8, 16, 32, 64
Reduced OR	`reduce_or`	4, 8, 16, 32, 64
Reduced XOR	`reduce_xor`	4, 8, 16, 32, 64
Adder/Subtractor	`add`	8, 16, 32, 64
Multiplier	`mul`	8, 16, 32, 64
Equal to	`eq`	8, 16, 32, 64
Less Than or Greater Than	`lgt`	8, 16, 32, 64
Divider	`div`	8, 16, 32, 64
Modulus	`mod`	8, 16, 32, 64

graph statistics are used as inputs to the Aggregation MLP to predict the physical characteristics of the input designs.

Table 1 lists all circuit graph node types and widths generated from the input designs to train and test the SNS prediction flow. Each unique pairing of *type* and *width* is a unique embedding of what we call a "vocabulary". The size of the vocabulary partly determines the difficulty in training the Circuitformer and the Aggregation MLP. Notice that the widths of the nodes are all power-of-two's and the maximal width is set at 64 bits. For all wiring connections that are not of the widths listed, we *round* the widths to the closest power-of-two. In addition, if a functional unit has multiple wiring connections and the connections are of varying widths, we use the maximal width.

Choosing the maximal wiring connection width and rounding to the closest power-of-two reduces the number of embedding vocabularies from around 1000 to 79 for the data set we used, greatly improves the training time of the downstream neural network models depicted in Figure 1 steps ③ and ④. Furthermore, a smaller set of embedding vocabulary allows better generalization of the Circuitformer when and if the training data is scarce. For example, a divider with a 17-bit width without rounding may never get trained when it is only seen once from the input data set. Whereas dividers with widths 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, and 23 are all considered an instance of the `div16` when rounding is employed and can be trained and inferred appropriately. Though the downstream model training time and accuracy improve with smaller vocabularies, there is some loss of information with respect to the original input design when rounding is used. Given that one of SNS's strengths is to explore large design spaces quickly and the final designs of interest will be synthesized at high fidelity using traditional synthesis tools, this loss of information should not affect the outcome of the designs chosen.

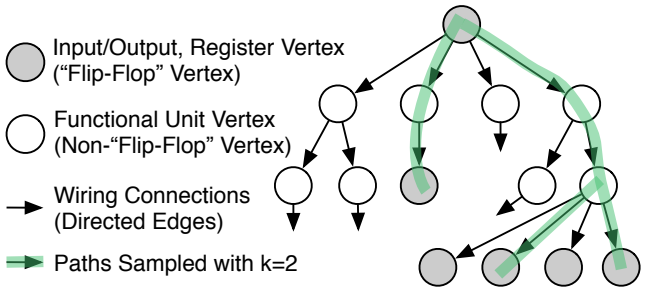


Figure 3: An Example of the Random Sampling Algorithm

3.2 Sampling Complete Circuit Paths from `GraphIR`

Recent work DeepWalk [24] analyzes social networks by using random path sampling over the entire undirected social network graph to obtain graph information and perform graph analysis. SNS is designed using a similar path-based approach, sampling random circuit paths over the entire directed circuit graph. In addition, SNS restricts the paths that can be sampled, leveraging domain knowledge of hardware design to reduce the large search space of all random paths. Instead of sampling paths in a design beginning and ending at any node, SNS samples paths beginning and ending with nodes that contain flip-flops. This restriction generates random paths that either begin or end with an input port, an output port, or a register (D-flip-flop) of some sort (e.g., pipeline register, local register). These paths essentially captures the "one-cycle behavior" of the design, making it possible for SNS to predict timing (i.e., the critical path) of circuit paths and entire designs. We call the paths that begin and end with vertices that contain flip-flops *complete circuit paths*. Figure 2 shows pictorially how a circuit graph in (b) is sampled exhaustively to form four complete circuit paths in (c). Each circuit path is represented with a sequence of vertices or nodes.

With the restriction of only sampling from complete circuit paths, the search space of the input design paths is reduced but still large. For example, for a RocketCore design [2] with an area of 0.07mm^2, exhaustively sampling all complete circuit paths results in 5.59 million paths. In reality, circuit paths do not need to be exhaustively sampled for us to predict the physical characteristics of the input design as lots of paths are similar and do not give us additional information. We develop a DFS-based (Depth-First-Search-based) algorithm to randomly sample complete circuit paths that are evenly distributed across the entire design shown in Algorithm 1. In this algorithm, we set a parameter k such that only $\lceil \frac{1}{k} \rceil$ of the successors (or at least one successor) of each vertex are traversed as part of the path. When $k = 1$, all complete circuit paths will be exhaustively sampled (as in Figure 2(c)). When $k = \infty$, only one successor will be traversed for each vertex. By choosing k, we can control how many paths we want to sample. Figure 3 shows an example of how paths can be sampled from a circuit graph when $k = 2$ so that half of the successors (or at least one successor) are traversed at each vertex. Note that each path begins and ends with a "flip-flop" vertex that is lightly shaded. For our SNS training (details in Section 4), we

Algorithm 1 Random Sampling of Complete Circuit Paths

r is a list to store paths sampled.
u is the vertex to start sampling from.
p is a list to keep track of unfinished sampled paths.
k is the parameter that controls how many paths are sampled.
Function Sample_Paths_From(u)
$r \leftarrow \emptyset$
if type(u) = dff $\vee$ type(u) = io **then**
  return r
else
  for all sample $\lceil \frac{1}{k} \rceil |u \rightarrow v|$ vertices v from $u \rightarrow v$ **do**
    $p \leftarrow$ Sample_Paths_From(v)
    $p \leftarrow$ prepend u to all paths in p
    $r \leftarrow r \cup p$
  end for
  return r
end if
end Function

choose $k = 5$ empirically as sampling more paths does not improve SNS model accuracy.

3.3 Predicting Circuit Path Physical Charateristics

The simplest and the most intuitive model for predicting physical characteristics of complete circuit paths is a linear regression model that takes counts of each type of vertices on a circuit path as inputs, because longer circuit paths lead to more vertices, and more vertices lead to larger area, power, and cycle time (lower frequency). However, a linear regression model is unable to infer based on the order or the placement of the vertices in a circuit path. Consider two complete circuit paths: [`io8, mul16, add16, dff16`] (shown in Figure 2(c)) and [`io8, add16, mul16, dff16`] where the multiplier and the adder positions are swapped. For the first circuit path, a traditional synthesizer like the Synopsys Design Compiler will infer that an adder followed by a multiplier can be synthesized as a MAC (multiply-accumulate) unit and produce a smaller area, power, and timing than synthesizing the second circuit path. Whereas a linear regression model will treat these two circuit paths as having exact same number and types of vertices, producing identical area, power, and timing predictions.

To improve the accuracy of our predictions, we need a model that takes into account the ordering and the placement of the vertices in a circuit path. Many natural language processing models are designed to learn the ordering, placement, and the relationships between words, making them good candidates. Transformer [33] is one of such models that uses the attention mechanism for language processing and inferring other sequential features such as sounds [13] and signals [40] beyond natural language processing. In our work, we augment the Transformer model to predict physical characteristics of complete circuit paths called the *Circuitformer*.

While the canonical Transformer model is too large and too slow for SNS, Circuitformer is designed for a much simpler task than understanding a natural language. Input hardware designs have much fewer vocabularies (shown in Table 1) than most natural languages not to mention that a language sentence conveys much more complicated semantics and meaning than a circuit path. Circuitformer is therefore designed by significantly reducing the size of the Transformer model into a light-weight model with two hidden layers and two attention heads. In addition, the embedding matrix, which stores the vector representation of each embedding, is much smaller in the Circuitformer. With these augmentations, we pruned the original Transformer model with 109 million parameters into a light-weight Circuitformer model with 1.4 million parameters. Table 2 shows the hyperparameters of the Circuitformer.

Table 2: Circuitformer and Transformer Hyperparameters

	Circuitformer	Transformer
Vocabulary Set Size	79	30522
Hidden Layers	2	12
Attention Heads	2	12
Embedding Vector Size	128	768
Maximum Input Size	512	512
Total #Parameters	1.4 M	109 M

3.4 Predicting Input Design Physical Charateristics

Having extracted higher-level features from individual circuit paths, we must now aggregate them in order to produce global inference results. For SNS, we use Multi-Level Perceptrons (MLPs) to aggregate path-level inferences to predict the area, timing and power for the entire design.

To reduce these path statistics into single values, we perform the following *reductions*:

3.4.1 Timing. The critical path of the design is the path that takes the longest time to execute. Therefore, given a set of paths, we output the *maximum* of the set.

3.4.2 Area. Because the path samples are evenly distributed, the expected area of the design is proportional to the sum of the area of each circuit path. Therefore, given a set of path areas, we output the *sum*.

3.4.3 Power. Because power is cumulative across circuit paths, we do the same as with circuit path's area and output the *sum*.

3.4.4 Power Gating. Register activity coefficients indicate how often a register is expected to switch states. Since the power consumption of a register is directly proportional to how often it switches states, knowledge of this information allows considerably more accurate results to be inferred. If the user provides clock gating behavior and activity coefficients for each register for their design, the path-based approach used by SNS enables the integration of this information to produce higher quality inferences. To do this, SNS simply *scales* the power of each path by the provided activity coefficient first and then *sums* them together.

For each target feature (area, power, and energy), the corresponding aggregation along with other graph statistics (e.g., functional unit counts, shown in Figure 2) are provided as inputs to an MLP

Table 3: Example Hardware Designs Selected

Processor Core	Rocket [2], Ariane [4], Sordor [1]
Peripheral Component	IceNet [1], RocketGPIO [1]
Machine Learning Acc.	Gemmini [10], NVDLA [7]
Vector Arithmetic	SIMD ALUs, Hwacha [19]
Signal Processing	FFT [25], Convolution
Cryptographic Arithmetic	AES [25], Sha3 [1]
Linear Algebra	GEMM [25], SPMV [25]
Sort	MergeSort [25], RadixSort [25]
Non-linear Function Approximation	Lookup Tables Piece-wise Approximation
Other	FP Unit [12], Stencil2D [25] Viterbi [25]

with three fully-connected layers each with 32 neurons, the output of which is the final inference for the desired feature.

4 THE DATASET GENERATION AND THE MODEL TRAINING OF SNS

In the SNS prediction pipeline, there are two deep learning models that have to be trained: the *Circuitformer* and the *Aggregation MLP*. In order to train the Circuitformer, a Generative Adversarial Network [11] (GAN) model needs to be trained also. In this section, we discuss how the training and testing datasets (a hardware design dataset and a circuit path dataset) are populated (i.e., collected or generated) as well as how these models are trained. Figure 4 shows the SNS training flow along with dataset dependencies.

4.1 Hardware Design Dataset Generation

To populate the input dataset using hardware designs, we first collect available, open-source Verilog designs. To avoid overfitting, we include hardware designs from various classes of applications and varying design complexities. We use many hardware designs from open-source projects and open-source collections such as Chipyard [1] and NVDLA [7]. MachSuite [25] also provides open-source implementations of commonly accelerated low-level kernels. However, MachSuite is a High Level Synthesis (HLS) benchmark suite and the benchmarks as they are written may not generate optimized Verilog designs. Thus, we re-implemented some of the kernels in MachSuite using the hardware description language Chisel [3] to produce high-quality and parameterizable Verilog designs. Table 3 lists a wide range of designs that are selected as part of our input hardware design dataset. Because some input designs we obtained are parametrizable, designs with different hardware parameters are generated whenever possible. In total, we obtained 41 hardware designs.

These designs are synthesized using the Synopsys Design Compiler [14] with the FreePDK 15nm library [21]. The synthesis results (i.e., area, timing, and power) combined with the Verilog source files of each design are compiled into a dataset we call the *Hardware Design Dataset* and its format is shown in Table 4. To access the generated dataset, the input design verilog files are compiled and turned into `GraphIR` on the fly before being used for the training and the testing of the SNS models. We use a 50-50 split for the training and testing sets. The models are evaluated in Section 5. Note that we avoid putting designs generated from the same parameterizable base design in both the training and the testing sets to ensure the fairness of our model evaluation.

Table 4: Format of the Hardware Design Dataset

Verilog Files	Timing	Area	Power
top.v, alu.v, func1.v ...	1000ps	10000um^2	1000mW
...	...	...	...

Table 5: Format of the Circuit Path Dataset

Paths	Timing	Area	Power
`[mul32, add32, ...]`	400ps	10um^2	0.01mW
...	...	...	...

4.2 Complete Circuit Path Dataset Generation

Recall that SNS uses predictions from the complete circuit paths and aggregates these results gradually into the prediction for the input design as a whole. To obtain predictions from these complete circuit paths, another dataset needs to be generated we call the *Circuit Path Dataset*. In essence, we want to populate an analogous dataset to the Hardware Design Dataset but instead of recording input design files, we record complete circuit paths in the form of a sequence of vertices or nodes that represent input/output ports, registers, or functional units; instead of synthesized timing, area, and power for the entire design, we obtain synthesized timing, area, and power for individual circuit paths. This dataset is used to train the Circuitformer; training details are described in the next subsection. The format of the database is shown in Table 5.

We populate the Circuit Path Dataset by randomly sampling from the hardware designs in the training set as described in Section 3.2. However, only few unique circuit paths can be sampled from a small hardware design training set (i.e., about 20 input designs). To adequately train the Circuitformer, we generate artificial but realistic circuit paths in the presence of input data scarcity employing the following two methods: Markov Chain and Sequence Generative Adversarial Nets (SeqGAN) [41].

4.2.1 Markov Chain Method. The construction of complete circuit paths consists of a sequence of vertices all with edges representing wiring connections. We first analyze the sampled circuit paths from the input design training set and compute a transition matrix; the transition matrix stores the conditional probability of the next vertex given the current vertex. The Markov Chain method then allows us to generate more circuit paths by using the transition matrix. The generated paths are unique but realistic as they are variants of paths directly sampled from real designs.

4.2.2 SeqGAN Method. The Markov Chain method, though simple and effective, does not work well for long complete circuit paths as the transition matrix only contains conditional probabilities of two adjacent vertices. To solve this problem, we use another method, SeqGAN, which is a Generative Adversarial Network for sequences. SeqGAN learns (and is trained with) the order of vertices from

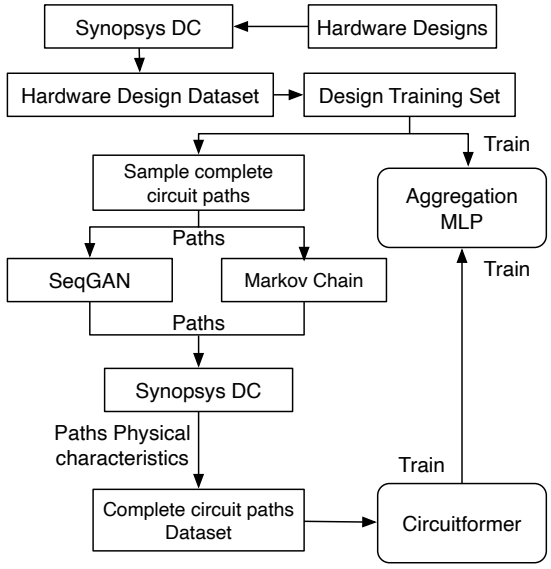


Figure 4: SNS Training Flow

Table 6: Training Hyper-parameters for SNS

Model	Optimizer	Batch Size	LR	Epochs
Circuitformer	Adam [15]	128	0.001	256
Aggregation MLP	SGD [30]	64	0.0001	10240
SeqGAN	Adam	2048	0.01	130

the paths directly sampled from real designs and then generates new unique, complete circuit paths that are similar to the ones it learned from. The advantage of the SeqGAN method over the Markov Chain method is that SeqGAN has a more holistic view of the paths, and therefore it generates more meaningful, longer complete circuit paths. However, it is still beneficial to include the Markov Chain method because it produces sequences that are less biased towards the training data and contain more noise than the SeqGAN method alone. Adding training data with higher noise and less bias to Circuitformer's training set results in a more robust and accurate model.

In total, we obtained 684 complete circuit paths from direct sampling of the hardware design training dataset. Using the methods described above, we generated an additional 4096 unique circuit paths (~1000 from the Markov Chain method and ~3000 from the SeqGAN method).

4.3 Model Implementation and Training

The Markov Chain model is implemented in-house without using any external libraries. The SeqGAN model is obtained from the SeqGAN's open-source project [41]. The dataset used to train the SeqGAN model are the complete circuit paths sampled from the training set of the Hardware Design Dataset. The Circuitformer model is implemented in PyTorch as an augmentation of the Transformer model using the Transformer library [36] from Hugging Face [35]

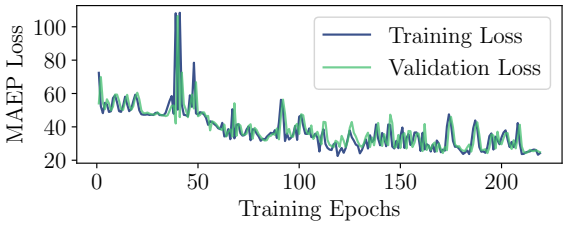


Figure 5: Circuitformer Training Loss vs. Validation Loss

and is trained using the Circuit Path Dataset. The Aggregation MLP model is implemented in PyTorch and trained using the training set of the Hardware Design Dataset as well as the path-specific predictions generated by the Circuitformer as shown in Figure 4. Table 6 shows the detailed training hyperparameters for each model. As the Circuitformer model is newly proposed, its training loss vs. validation loss are shown in Figure 5.

5 EVALUATION

In this section, we first present the evaluation of SNS itself, comparing the accuracy of the predicted area, power, and timing to traditional synthesis tool results using Synopsys Design Compiler (Synopsys DC) as our baseline. We also compare the wall clock time to run the SNS predictions with the baseline to assess the performance of SNS. We then present two case studies to highlight SNS's capability, scalability, and validity using a general-purpose processor core (RISC-V BOOM) case study and a classic machine learning accelerator (DianNao) case study.

5.1 Metrics for Evaluation

We use the following two metrics for evaluating the accuracy of SNS: Mean Absolute Error Percentage (MAEP) and Root Relative Square Error (RRSE). MAEP is an intuitive and tangible metric that expresses the mean percentage by which a prediction differs from the ground truth. However, this metric does not consider the scale and distribution of the prediction space, creating bias against models predicting features whose range is larger. To remedy this, RRSE scales the root mean square error by the variance of the ground truth. This results in a metric that is invariant in the size of the range of the predicted feature.

In our evaluation, RRSE is used as the main metric for comparing predictions generated by SNS and Synopsys DC (i.e., our ground truth), but MAEP is still provided because it provides a more intuitive basis for comparison.

5.2 SNS Prediction Accuracy

Using the SNS models, training flow, and the generated Hardware Design Dataset described in Section 4, we train and evaluate using the training set size of 50% and use the other 50% as the testing set. To evaluate SNS accuracy, we use the prediction flow depicted in Section 3 and predict the physical characteristics of input designs in the testing set. The prediction accuracy results are obtained in a 2-fold cross-validated fashion where the dataset is first split into part A and part B, each containing 50% of the designs. Then part

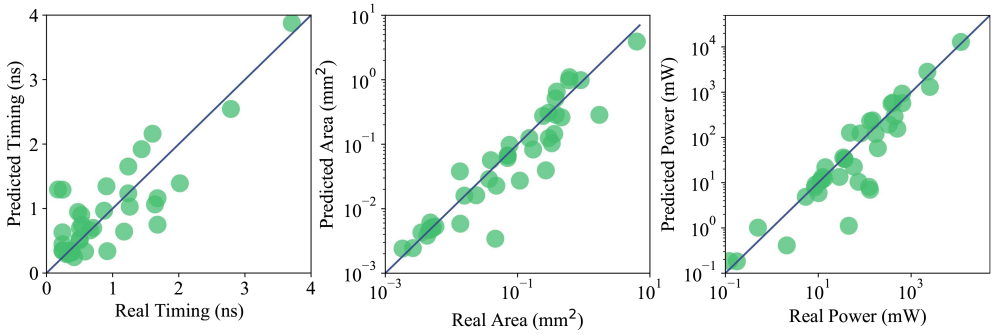


Figure 6: SNS Prediction Accuracy to Predict Physical Characteristics of the Input Designs

Table 7: Evaluation Accuracy (Lower Better)

SNS Prediction Error	Training Set %		D-SAGE
	50%	30%	
Timing RRSE	0.67	0.82	0.83
Power RRSE	0.60	1.02	-
Area RRSE	0.22	0.26	-
Timing MAEP	38.00%	61.46%	-
Power MAEP	48.72%	71.35%	-
Area MAEP	54.57%	52.02%	-

A is evaluated using the model trained on part B while part B is evaluated using the model trained on part A.

We compare the predictions against synthesis results generated by Synopsys DC as our baseline. The distribution of the 2-fold cross-validated area, power, and timing predictions are depicted pictorially in Figure 6. Y-axis plots the predicted values while the X-axis plots the baseline values; the closer the design points are from the diagonal lines, the more accurate the predictions are. We make two observations from these three plots. First, SNS generally produces acceptable synthesis results with few hard-to-predict designs. For the use cases that we envision SNS to be the most useful, such as performing large design space explorations, these predictions will suffice in allowing the hardware developers to narrow large design spaces and select desired designs. The selected designs can then be synthesized, placed, and routed using traditional synthesis toolsuite for high fidelity results before fabrication. Second, the area and power axes are in log-scale, showing the wide range of design points that are tested. For prior works [42, 44], only small designs can be predicted. PRIMAL [44] and GRANNITE [42] can predict designs up to 100k gates and 50k gates respectively while SNS can predict designs up to 18 million gates.

To assess the robustness of SNS when training data is even more scarce, we perform another experiment using 30% of the Hardware Design Dataset as the training set and the other 70% as the testing set. The RRSE and MAEP results of both the 50% and the 30% training sets are detailed in Table 7. As expected, with a smaller number of input designs as the training set, the model is not as accurate as the ones trained on more input designs.

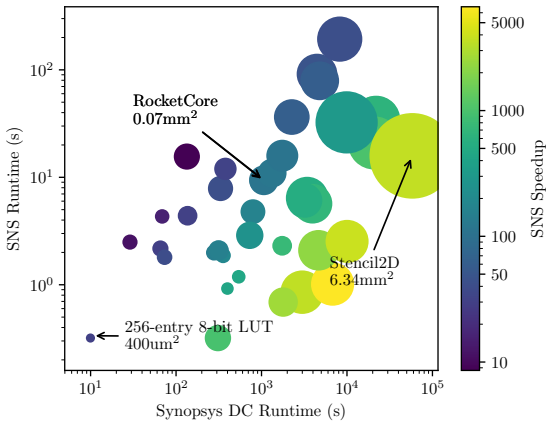


Figure 7: SNS runtime vs. Synopsys DC runtime

5.3 Comparison with Related Works

Out of all the related works, D-SAGE [32] is probably the most closely related to our work and is also state-of-the-art. D-SAGE is a GNN-based operation delay predictor for HLS designs on FPGA platforms, whereas SNS is a path-based and Transformer-based synthesis predictor for Verilog designs. SNS predicts for area and power while D-SAGE does not; D-SAGE does edge classification while SNS does not. Both D-SAGE and SNS predict for timing of hardware designs. D-SAGE achieves a timing prediction accuracy with RRSE = 0.82; SNS is able to achieve the same level of accuracy compared to D-SAGE using just 30% of the input designs as the training set (Timing RRSE = 0.83), and outperforms D-SAGE (Timing RRSE = 0.68; lower better) with 50% of the input designs as the training set. In addition to the quantitative comparison with D-SAGE, Table 8 shows a qualitative comparison of SNS with other recent related works, highlighting the capabilities of SNS.

5.4 SNS Performance Evaluation

SNS is designed to be an ultrafast predictor. For the 41 designs in our dataset, we measure the runtime of SNS and the runtime of Synopsys DC (both using wall clock time) to obtain the physical characteristics of the hardware designs using a medium-sized server platform with configurations shown in Table 9. The comparison of

Table 8: A Qualitative Comparison with Related Works

	D-SAGE [32]	**Aladdin** [27]	**MAESTRO** [18]	**ParaGraph** [26]	**APOLLO** [39]	**SNS**
Timing Prediction	Yes	Yes	No	Yes	No	Yes
Area Prediction	No	Yes	Yes	Yes	No	Yes
Power Prediction	No	Yes	Yes	Yes	Yes	Yes
ASIC Design Prediction	No	Yes	Yes	Yes	Yes	Yes
FPGA Design Prediction	Yes	No	No	No	No	No
Support General Purpose Designs	Yes	No	No	No	No	Yes
Support Large Designs (>1M gates)	No	Yes	Yes	No	Yes	Yes
No Human Intervention	Yes	No	No	No	Yes	Yes

Table 9: Configuration of Benchmark Platforms

	Server Platform	**Desktop Platform**
Host Processor	2 Intel Xeon Gold 6252 48C/96T@2.10GHz	Intel Core i9 11900 8C/16T@2.5GHz
Memory	8 64GB 2933MHz	2 16GB 2667MHz
Storage	1TB SATA SSD	256GB SATA SSD
OS	Ubuntu 18.04LTS	Ubuntu 18.04LTS

the runtime between SNS and Synopsys DC is shown in Figure 7 with the SNS runtime on the Y-axis and the Synopsys DC runtime on the X-axis. Each design point is represented as a circle, with area (and complexity) of the design proportional to the size of the circle. The color shades of the circle represents the SNS speedup w.r.t. the baseline; darker slower and lighter faster. We highlight three example designs along with their areas: a small lookup table, an in-order processor core without caches, and a large 16-core accelerator for processing floating point stencil 2D calculations.

We make two observations from this figure. First, SNS achieves a speedup of up to three orders of magnitude compared to Synopsys DC to predict area, power, and timing of the input designs. Second, for larger designs that take longer to synthesize using Synopsys DC, SNS shows an even larger speedup compared to the small designs. The speedup of SNS over Synopsys DC on average is 760×.

In addition to this experiment, we consider the use case of running SNS on a desktop that is much less powerful than the server (configuration in Table 9). We benchmarked SNS on the desktop platform and compared it with the Synopsys DC runtime on the server. The result shows that SNS still achieves an average speedup of 574×.

5.5 Usage Model

System Verilog, Verilog HDL, and VHDL are directly supported by SNS. Other parameterizable hardware description languages such as Chisel [3] and PyMTL [20] can be used as hardware generators that produce synthesizable Verilog, and the output Verilog can then be used as inputs to SNS. HLS designs can also be indirectly supported by using tools such as Bambu [8] to generate the synthesizable HDL.

Besides supporting designs written in a broad range of hardware description languages as well as HLS designs, SNS can be used to perform large scale design space explorations. When doing so, a parameterizable design is first compiled with combinations of design parameters to form fixed RTL designs. SNS then predicts the physical characteristics of all the generated RTL designs from the previous step. Finally, the physical characteristics of the designs are analyzed to find the best parameter choices. We demonstrate two such case studies in the following two subsections.

5.6 BOOM Case Study

When developing a general-purpose processor core, especially one with reasonable complexity, such as the RISC-V out-of-order core BOOM [43], there are several tens of parameters that would result in hundreds to thousands of possible designs. To select the best designs that meet either a performance goal or a power and area efficiency goal, performing a design space exploration using all possible parameters can be prohibitively expensive. For example, synthesizing a typical BOOM core takes around 2.5 hours to complete using Synopsys DC on a server consuming 16 cores and 45 GB of memory.

One of the strengths of SNS is its speed and a natural use case is to enable a large and high-dimensional design space exploration (DSE) in order to select the desired Pareto designs during the hardware development process. We conduct such an experiment using the BOOM design parameters listed in Table 10. For this DSE of 2592 designs, SNS took 2.1 hours to complete the synthesis predictions using the server in Table 9. If we were to conduct the same experiment using Synopsys DC, it would have taken around 45 days.

To verify the accuracy of SNS predictions, we randomly sampled 20 design points from the 2592 designs and synthesized them using Synopsys DC to compare the synthesized results with the predicted results. SNS is able to achieve MAEPs for area, power, and timing of 12.58%, 29.61%, and 19.78% respectively, demonstrating the accuracy of SNS.

We obtain the performance of these designs by running CoreMark [9] using the cycle accurate simulator provided by Chipyard for each of the BOOM cores in the design set. By scaling each CoreMark score with the frequency predicted by SNS and incorporating the area and power predicted by SNS, we obtain Figures 8 that plotted performance vs. power and performance vs. area respectively. We choose three designs on the Pareto frontier: The highest performance design (`HighPerf`), the most power-efficient design (`PowerEff`), and the most area-efficient design (`AreaEff`). The hardware parameters for these three designs are shown in Table 11.

Table 10: Boom DSE Hyper parameters

Parameter Name	Possible Values	Count
Branch Predictor	TAGE-L, Boom2, Alpha21264	3
Core Width	1, 2, 3, 4	4
Memory Ports	1, 2	2
Instruction Fetch Width	4, 8	2
ROB Size	32, 64, 96	3
Physical Integer Registers	52, 80, 100	3
Issue Slots	8, 16, 32	3
L1 Data Cache Ways	4, 8	2
	# of Combinations:	2592

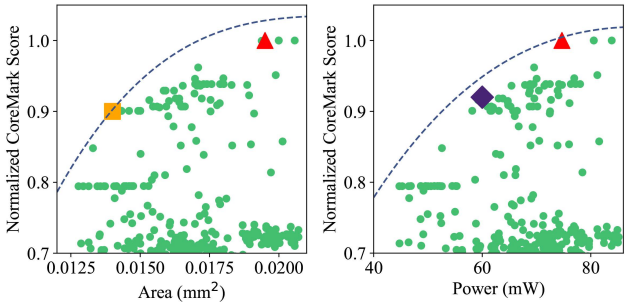


Figure 8: Boom DSE Result: The `HighPerf` design is plotted using a triangle shape, the `PowerEff` design is plotted using a diamond shape, and the `AreaEff` design is plotted using a square shape. The coremark scores are linearly normalized such that the fastest core in our design has a score of one.

Based on the DSE result, we can make several observations about the BOOM core design. First, we find that few designs have similar performance compared to the `HighPerf` (i.e., the designs directly to the right of `HighPerf`) but consume more power and more area. These designs have 32 issue slots rather than the 16 slots in the `HighPerf` design. This implies that having more issue slots in the 4-wide core will not provide a speedup because the core is bottlenecked by the decoder and not the issue queue. Second, although the `PowerEff` and the `AreaEff` designs have fewer issue slots, many fewer integer registers, and much smaller ROBs than the `HighPerf` design, they are only marginally (less than 10%) slower. This shows that general-purpose single-core processors can suffer from diminishing performance returns from allocating too many resources. Finally, all designs close to the Pareto frontier have only a single memory port since CoreMark is not a memory intensive benchmark and therefore is not bottlenecked by memory throughput.

5.7 DianNao Case Study

DianNao [6] is a neural network accelerator specialized for CNN inference. As shown in Figure 9, the DianNao pipeline is divided into three stages, NFU-1, NFU-2 and NFU-3 where T_n defines the shape and scale of each stage:

Table 11: Boom Selected configurations

Parameter Name	`BestPerf`	`PowerEff`	`AreaEff`
Branch Predictor	TAGE	Alpha21264	TAGE
Core Width	4	4	2
Memory Ports	1	1	1
Fetch Width	8	8	4
ROB Size	64	32	32
Integer Registers	100	52	52
Issue Slots	16	8	8
L1 Data Cache Ways	8	8	4

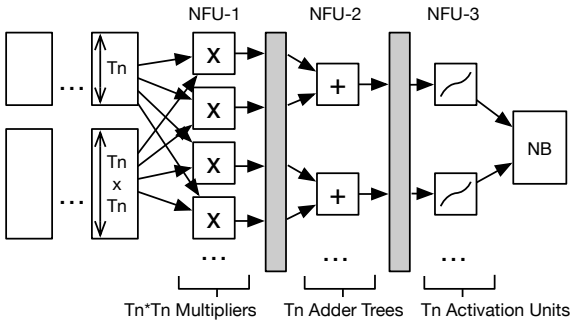


Figure 9: Abbreviated Reproduced Diagram for DianNao

- NFU-1: Composed of $T_n \times T_n$ multipliers where T_n also stands for the maximum number of input neurons the DianNao pipeline can process each cycle. NFU-1 is responsible for all multiplications in fully-connected layers and convolution layers.
- NFU-2: Composed of T_n adder trees each with T_n inputs. This layer is responsible for the summation of multiplication results.
- NFU-3: Composed of T_n activation units where the activation function is approximated by linear piece-wise functions with break-points, offset and slope stored in a lookup table. NFU-3 is responsible for the activation of neurons.

In this case study, we want to answer the following questions quantitatively: 1) Can SNS predict the original DianNao synthesis results? 2) How will different values for T_n affect the efficiency of the accelerators? 3) How will different datatypes affect the hardware complexity, efficiency, and model accuracy?

To answer these questions, we implement our version of DianNao using Chisel [3]. We make it parameterizable and support floating point operations in order to perform a design space exploration on T_n and the input datatypes. We build a cycle accurate performance model for DianNao which generates clock-gating activity coefficients for each register in the hardware. We evaluate the accuracy of DianNao hardware on the CIFAR-10 [16] dataset using AlexNet [17], which we train using the training set provided by CIFAR-10.

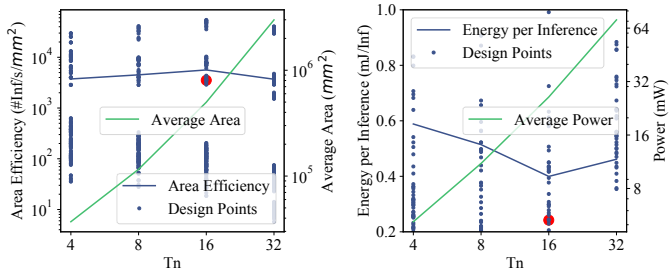
To predict DianNao's synthesis results, we implement the DianNao design with the parameters provided in the published paper

Table 12: SNS 's Synthesis Prediction for DianNao

	Power (mW)	**Area** (mm^2)	**Timing** (ns)
Synthesis Result (65nm)	132	0.846563	1.02
Scaled Result (15nm)	65.90	0.097302	0.33
SNS Prediction (15nm)	59.26	0.070269	0.36

Table 13: DianNao DSE Design Parameters

Parameter Name	**Possible Values**	**Count**
T_n	4, 8, 16, 32	4
Datatype	int8, int16, fp16, bf16, tf32, fp32	6
Pipelines stages (name:# of stages)	3 (NFU-1:1, NFU-2:1, NFU-3:1), 8 (NFU-1:3, NFU-2:2, NFU-3:3)	2
Reduction Width	4, 8, 16	3
Activation Entries	2, 4, 8, 16	4
	Total #Combinations	576



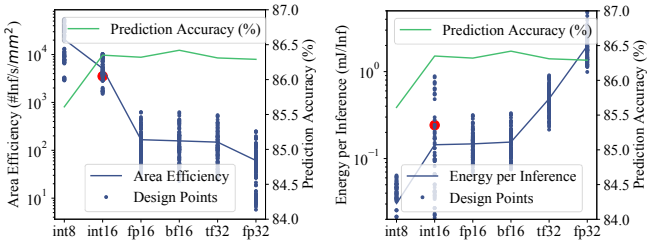
(a) T_n (#Input Neurons) vs. Area Efficiency vs. Area **(b) T_n (#Input Neurons) vs. Power Efficiency vs. Power**

Figure 10: Design Space Exploration on T_n, the original DianNao design is plotted as the red dot in the figures

and use SNS's prediction flow. Using the activity coefficients generated from the performance model, we can predict DianNao's power consumption with power gating. SNS is able to predict the area, power, and timing of DianNao with the error of 27.8%, 10.1% and 9.1% respectively, shown in Table 12. Note that the original synthesis results of the DianNao are in 65nm (first row of Table 12) which have to be scaled [28] to the technology size that SNS uses (15nm).

To perform our design space exploration on T_n and the input datatype, we generated 576 DianNao designs using all possible combinations of parameters shown in Table 13, and used SNS to predict the power, area, and timing for these designs. The prediction process takes 809 seconds running on the server described in Table 9.

We plot the power efficiency and the area efficiency in Figures 10(a) and (b) for different T_n. Design points and effciencies are plotted against the primary Y-axis while the area and power are plotted against the secondary Y-axis. Figure 10(a) shows that the area of the designs increases as T_n goes from 4 to 32 and the area efficiency (which is defined as the inference throughput per unit



(a) Datatype vs. CIFAR-10 Accuracy vs. Area Efficiency **(b) Datatype vs. CIFAR-10 Accuracy vs. Power Efficiency**

Figure 11: Design Space Exploration on different datatypes, the original DianNao design is plotted as the red dot in the figures

area) is at the maximum when $T_n = 16$. Figure 10(b) shows that the power of the designs also increases for larger T_n and the energy per inference is also the lowest for $T_n = 16$ (meaning designs with $T_n = 16$ have the best power efficiency compared to designs with other T_n). This analysis shows that $T_n = 16$ is an optimum for both area and power efficiency which explains why the DianNao paper chooses $T_n = 16$ for their design.

Hardware complexity vs. accuracy is another important trade-off to consider when designing neural network accelerators. Figures 11(a) and (b) show that using a datatype with less expensive operations greatly increases the efficiency of a design in terms of both area and power. Especially for our specific task of CIFAR-10 image classification, going beyond Int16 does not provide us with any appreciation in accuracy. This suggests that using Int16 is optimal and explains why the original DianNao paper chooses Int16.

6 RELATED WORK

With the increasing complexity of computer chip designs, it is becoming prohibitively slow to use conventional techniques such as gate-level synthesis and cycle-accurate simulation to model large systems. To efficiently and accurately model large-scale systems, machine learning techniques have been used to predict modeling results, often leading to orders of magnitude faster runtime.

Traditional Machine Learning Models. Traditional Machine Learning models have been used to predict various computer system characteristics, including the power, area (resource usage), and timing of particular hardware designs. Barboza et al. [5] proposed using an ensemble of models including Lasso, Artificial Neural Networks (ANNs), and Random Forests to reduce the pessimism of timing predictions provided by commercial tools. Pyramid [22] proposed a machine learning framework that estimates the resource usage of an HLS design where they tested and compared the prediction accuracies of many ML models including Linear Regression, ANNs, Supporting Vector Machines (SVMs), and Random Forests. Apollo [39] designed a linear model to model the power of a microprocessor executing different workloads with an R^2 >0.95 accuracy and showed that the simulation time was reduced from months to minutes.

Although traditional machine learning models are fast and robust, it is unable to perform complicated inferences such as inferring the physical characteristics of an arbitrary hardware design based solely on the input HDL source codes as we demonstrated in this work.

Convolutional Neural Networks for Power Estimation. Convolutional Neural Networks (CNNs) can predict the power of integrated circuits with reasonably good accuracy. PRIMAL [44] was able to predict the cycle-accurate power of RTL designs 50× faster than gate-level power analysis tools with the help of CNNs. Xie et al. improved the results further by introducing Max-CNN [38] [37], increasing the accuracy, and making the model transferable to different designs. CNNs can be utilized to predict the power of a hardware design well, however, it is not able to infer path-level characteristics such as critical path timing as SNS can.

Graph Neural Networks for Circuits. Using machine Learning techniques to infer circuit characteristics by designing appropriate feature extraction mechanism is a particularly difficult task. Both CNNs and traditional machine learning models require carefully designed feature extraction mechanisms. In comparison, Graph Neural Networks (GNNs) can work on graph representations of circuits directly without the need to do feature extraction. ParaGraph [26] proposed a GNN based model to predict device parameters and layout parasitics. GRANNITE [42] uses GNNs to predict the runtime power of a circuit, achieving an 18.7× speedup over gate-level simulations with less than 5.5% error. D-SAGE [32] is a customized GraphSage model specializing in predicting the operation delay of HLS designs.

GNNs can predict various physical characteristics of hardware designs with high accuracy, however, it is slow and does not scale well to very large circuits. We demonstrated the performance as well as the scalability of SNS in this work, showing that SNS performs on average 760× better than Synopsys DC and is able to infer very large circuits up to 18 million gates.

A Path-based Approach from DeepWalk. DeepWalk [24] is an alternative approach to graph analysis that, instead of using GNNs and message passing mechanisms, performs random walks of the graph. As a result, DeepWalk is much faster, and is able to scale to large graphs whereas GNNs cannot. As we have demonstrated in this work, this path-based approach proposed by DeepWalk generalizes well and can be expanded and applied towards circuit analysis.

Non-Machine Learning Models for Reducing Synthesis Result Feedback Delay. Machine Learning methods are not the only possible way to reduce the synthesis result feedback cycle. AutoAx [23] proposed using approximated components to synthesize the HDL files, which makes the synthesis process 162× faster. Aladdin [27] proposed a framework to estimate the performance, power, and area of the accelerators within an SoC platform pre-RTL using non-learning-based models. It reduces the cycle of knowing the synthesis result from tens of hours to minutes while achieving high accuracy. Both of these methods cannot predict arbitrary hardware designs with similar post-RTL, post-synthesis accuracy without human intervention.

7 CONCLUSION

Rapid development and refinement of hardware designs require that synthesis results can be obtained quickly. However, synthesizing a small design, for example, a 32×32 systolic array with an area of approximately 2.5mm^2, takes one full day to complete using traditional synthesis toolchains such as the Synposys Design Compiler. With growing design sizes and therefore growing synthesis delays, modern hardware design toolchains have had to make compromises such as adopting modular design methodologies, which can have negative effects on the overall quality of the design.

To remedy these challenges, we have shown how SNS uses modern machine learning techniques to predict synthesis results with high accuracy. While it does not provide a synthesized netlist, it provides many of the design level properties (i.e., timing, power, and area estimates) necessary to inform hardware developers in their design process.

While prior work [22, 32, 38, 42] used representations of the entire circuit graph to perform inference, SNS takes inspiration from Transformer networks [33] and DeepWalk [24] and takes a different approach. SNS samples paths from the input design and performs inference on them using a novel, lightweight Transformer model called *CircuitFormer* to accurately deduce local properties of the overall design. These local properties are then aggregated and refined using Multi-Layer Perceptrons to produce inferences on global properties of the circuit design, providing an average 760× speedup over traditional synthesis toolchains with an average RSSE error of 0.4998.

We demonstrate the performance and accuracy of this model by performing exhaustive design space explorations on a general purpose out-of-order RISC-V processor BOOM [43] and on a classical machine learning accelerator, DianNao [6]. Our results showed that by using SNS, high-dimension design space explorations can now be performed in a reasonable amount of time and can properly inform the developer in selecting the optimal parameters for their design.

ACKNOWLEDGMENTS

We thank the reviewers for their insightful comments. This work was supported in part by an National Science Foundation CAREER award CCF-2045974, 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, et al. 2020. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro* 40, 4 (2020), 10–21.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012.* 1212–1221. https://doi.org/10.1145/2228360.2228584

[4] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. OpenPiton+ Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 1–6.

[5] Erick Carvajal Barboza, Nishchal Shukla, Yiran Chen, and Jiang Hu. 2019. Machine learning-based pre-routing timing prediction with reduced pessimism. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[6] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[7] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim. *CoRR* abs/1903.06495 (2019). arXiv:1903.06495 http://arxiv.org/abs/1903.06495

[8] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1327–1330. https://doi.org/10.1109/DAC18074.2021.9586110

[9] Shay Gal-On and Markus Levy. 2012. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* (2012).

[10] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. 2019. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925* 3 (2019).

[11] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Networks. arXiv:1406.2661 [stat.ML]

[12] John Hauser. 2022. *Berkeley Hardfloat.* Retrieved April 18, 2022 from http://www.jhauser.us/arithmetic/HardFloat.html

[13] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. 2018. Music transformer. *arXiv preprint arXiv:1809.04281* (2018).

[14] Synopsys Inc. 2014. *Design Compiler Graphical: Create a Better Starting Point for Faster Physical Implementation.* Technical Report. 700 East Middlefield Road, Mountain View, CA 94043.

[15] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/ARXIV.1412.6980

[16] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[18] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro* 40, 3 (2020), 20–29.

[19] Yunsup Lee, Colin Schmidt, Albert Ou, Andrew Waterman, and Krste Asanović. 2015. The Hwacha vector-fetch architecture manual, version 3.8. 1. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[20] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 280–292. https://doi.org/10.1109/MICRO.2014.50

[21] Mayler Martins, Jody Maick Matos, Renato P. Ribas, André Reis, Guilherme Schlinker, Lucio Rech, and Jens Michelsen. 2015. Open Cell Library in 15Nm FreePDK Technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design* (Monterey, California, USA) *(ISPD '15)*. ACM, New York, NY, USA, 171–178. https://doi.org/10.1145/2717764.2717783

[22] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Houman Homayoun, and Setareh Rafatirad. 2019. Pyramid: Machine Learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 397–403. https://doi.org/10.1109/FPL.2019.00069

[23] Vojtech Mrazek, Muhammad Abdullah Hanif, Zdenek Vasicek, Lukas Sekanina, and Muhammad Shafique. 2019. autoax: An automatic design space exploration and circuit building methodology utilizing libraries of approximate components. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[24] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[25] 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)*. IEEE, 110–119.

[26] Haoxing Ren, George F Kokai, Walker J Turner, and Ting-Sheng Ku. 2020. ParaGraph: Layout parasitics and device parameter prediction using graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[27] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 97–108.

[28] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[29] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40, 2 (2020), 45–52. https://doi.org/10.1109/MM.2020.2974217

[30] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*. PMLR, 1139–1147.

[31] Martin Svedin, Steven WD Chien, Gibson Chikafa, Niclas Jansson, and Artur Podobas. 2021. Benchmarking the Nvidia GPU Lineage: From Early K80 to Modern A100 with Asynchronous Memory Transfers. In *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. 1–6.

[32] 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*. 1–9.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[34] Clifford Wolf. 2016. Yosys open synthesis suite. https://yosyshq.net/yosys

[35] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2019. HuggingFace's Transformers: State-of-the-art Natural Language Processing. https://doi.org/10.48550/ARXIV.1910.03771

[36] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[37] Zhiyao Xie, Hai Li, Xiaoqing Xu, Jiang Hu, and Yiran Chen. 2020. Fast IR drop estimation with machine learning. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–8.

[38] Zhiyao Xie, Haoxing Ren, Brucek Khailany, Ye Sheng, Santosh Santosh, Jiang Hu, and Yiran Chen. 2020. PowerNet: Transferable dynamic IR drop estimation via maximum convolutional neural network. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 13–18.

[39] Zhiyao Xie, Xiaoqing Xu, Matt Walker, Joshua Knebel, Kumaraguru Palaniswamy, Nicolas Hebert, Jiang Hu, Huanrui Yang, Yiran Chen, and Shidhartha Das. 2021. APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–14.

[40] Genshen Yan, Shen Liang, Yanchun Zhang, and Fan Liu. 2019. Fusing transformer model with temporal features for ECG heartbeat classification. In *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 898–905.

[41] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.

[42] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. 2020. GRANNITE: Graph neural network inference for transferable power estimation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[43] 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*.

[44] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. 2019. PRIMAL: Power inference using machine learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.