# METRICS2.1 and Flow Tuning in the IEEE CEDA Robust Design Flow and OpenROAD

*ICCAD Special Session Paper*

Jinwook Jung*, Andrew B. Kahng, Seungwon Kim and Ravi Varadarajan

UC San Diego, La Jolla, CA, USA

*IBM T. J. Watson Research Center, Yorktown Heights, NY, USA

***Abstract*—In today's RTL-to-GDS flow domain, there is a lack of standards for reporting of design and tool metrics. Moreover, each tool or engine has its own set of parameters that can change outcomes and trade off PPA and other metrics. Thus, the study and optimization of impacts of parameter settings across the entire RTL-to-GDS tool chain has been largely ad hoc. In this paper, we first describe METRICS2.1, a proposed standard for RTL-to-GDS design tool and flow metrics. We then describe how data collected using a METRICS2.1 realization can be analyzed to give insight into flow tuning and fields of use for PPA optimization. Last, we discuss hyperparameter autotuning in the RTL-to-GDS flow. We present *AutoTuner*, which uses derivative-free optimization to handle challenges of non-differentiability and many local minima. An open repository based on METRICS2.1 has been established for sharing of reproducible, standardized metrics data, along with example implemented applications, to support academic and industrial research on machine learning for tool/flow tuning.**

## I. Introduction

In the RTL-to-GDS domain, EDA tools have rapidly evolved to meet the complex optimization needs of advanced IC design in leading-edge manufacturing technologies. Recent years have seen EDA researchers actively apply machine learning (ML) techniques to enhance quality of results (QoR) and automation in numerous flow stages. However, the lack of an open, standardized metrics format hampers progress by necessitating ad hoc approaches to design, tool and flow data collection. Moreover, fragmented metrics formats block sharing of generated models for machine learning, and make it difficult to reproduce results.

A METRICS 1.0 infrastructure for the EDA and IC industries was proposed in the late 1990s [11], [16] to measure all design activity, mine all design process data, predict tool outcomes, find sweet spots or field of use for tools, and perform design-specific tuning of tool parameters. Nearly 20 years later, METRICS 2.0 revisited metrics and proposed an updated architecture for collection and sharing of data for machine learning applications [13]. During this period, commercial EDA vendors also developed proprietary ways of reporting metrics in tool logfiles and custom reports. Major platforms offer a unified reporting mechanism to capture both tool metrics and tool runtime parameters. However, commercial platforms are closed, and there is a lack of consistency across the different tools and companies.

To improve QoR outcomes for complex tools and flows, blackbox hyperparameter tuning, or *autotuning*, has been actively studied in recent years. Autotuning studies in the RTL-to-GDS flow domain have leveraged a variety of search algorithms [1], [21], [25], [26], [27]. However, lack of unified naming and format for metrics incorporated into reward functions limits the potential to apply a single framework across multiple EDA tools. The fact that many frameworks are not open, or have search algorithms strongly tied to the framework, hinders progress by tool users and academic researchers.

In this paper, we first propose *METRICS2.1* as a new standard for metrics collection and design process recording. The goals of METRICS2.1 are (i) to provide a standardized format of metrics data, and (ii) to define a robust structure for large metrics archives.[1] The METRICS2.1 data format is generic and flexible, so that it can evolve continuously and support user customization. We also present an archive of large-scale designs of experiments executed using the open-source OpenROAD toolchain [5], [35], [18], together with flow configuration information that enables reproducibility, as a basis for future ML applications. Leveraging the METRICS2.1 infrastructure, we further propose an open-source framework for RTL-to-GDS flow parameter tuning. The proposed tuning framework can support various EDA tool flows thanks to METRICS2.1, while allowing users to easily choose parameter search algorithms. We describe an exemplary objective function that allows weighting of power, performance and area QoR elements according to user requirements. The main contributions of our work are summarized as follows.

- We propose a new METRICS2.1 infrastructure that enables standardized metrics collection and design process recording for tool improvement and machine learning applications. We provide archives obtained from thousands of RTL-to-GDS flow executions. We also share analyses and ML applications in the form of Jupyter notebooks as a starting point for new researchers in the area of ML for CAD/EDA.
- We describe a fully open hyperparameter autotuning framework, *AutoTuner*, for the RTL-to-GDS tool chain. The proposed framework supports synchronous/asynchronous parallelization, flexible switching between search algorithms, and metrics collection using METRICS2.1 for reward function evaluation.
- We present two case studies of autotuning on open designs implemented in SkyWater $130nm$ (SKY130HD)

[1]METRICS2.1 is supported by a new initiative of the IEEE CEDA Design Automation Technical Committee (DATC), and is now incorporated in the DATC's Robust Design Flow (RDF).

and ASAP $7nm$ (ASAP7) technologies. Significant QoR improvements are achieved using objective functions that target elements of PPA quality measures. We also show tradeoffs between exploration and exploitation seen in walltime and autotuning results, when asynchronous parallelization is applied.

The rest of this paper is organized as follows. Section II introduces the METRICS2.1 standard and infrastructure, including a proposed hierarchical JSON format. Section III describes METRICS2.1 data sharing infrastructure and example ML analyses, demonstrated with Jupyter notebooks. Section IV describes our open-source EDA flow autotuning framework leveraging METRICS2.1. Section V presents case studies and experimental results using AutoTuner. Section VI summarizes and concludes the paper.

## II. METRICS2.1

METRICS2.1 aims for simplicity and extensibility, with clearly-defined syntax and semantics to enable future addition of new metrics. A guiding precept is that (i) any desired measurement must map to a unique METRICS2.1 metric; and (ii) any METRICS2.1 metric must map to a unique interpretation as a measurement that should be intuitively obvious to mainstream users. This two-way mapping is crucial to avoid a “Tower of Babel” situation. Moreover, in a typical EDA design flow, the value of a specific metric changes throughout the flow. For example, the number of instances in the design changes as the design goes through various *stages* such as synthesis, placement, optimization, CTS, etc. Hence, it is important to capture the same metric at different stages of the design flow. This is essential to monitor the design PPA across flow stages, and to perform trend analysis of specific metrics over the course of the design flow. With this in mind, we establish key decisions and the following naming conventions for METRICS2.1.

- Metrics are organized hierarchically into *stages* or *snapshots.*
- METRICS2.1 has predefined stages for a typical design flow, but a user can also add custom snapshots that essentially correspond to user-defined stages.
- Each metric belongs to a predefined metrics *category*.
- Each metric has a predefined metric *name* and an optional predefined metrics *name modifier*.
- A metric name and the optional metric name modifier implicitly define the units of the metric, and can be used as is to represent a valid metric. Example: “design__instance__count”, “timing__setup__wns”, “power__internal”.
- A metric can also have optional *classification modifiers* to further classify the metric. The classification modifiers are either by *type* or *structure*.
- When we have both a type classification modifier and a structure classification modifier, the type classification modifier appears first, followed by the structure classification modifier. METRICS2.1 documentation establishes a precedence order for classification modifiers such that, e.g., we cannot have both “double-height regular-VT” and “regular-VT double-height” modifiers.

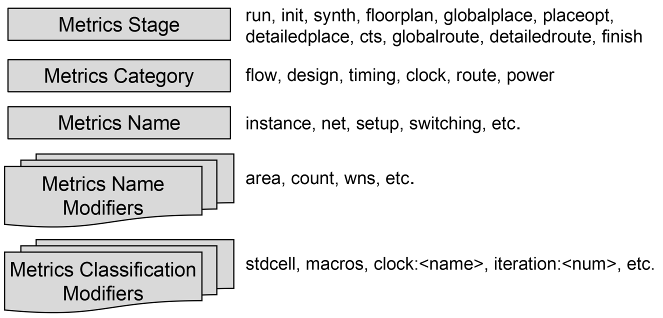

Metrics Stage
run, init, synth, floorplan, globalplace, placeopt, detailedplace, cts, globalroute, detailedroute, finish
Metrics Category
flow, design, timing, clock, route, power
Metrics Name
instance, net, setup, switching, etc.
Metrics Name Modifiers
area, count, wns, etc.
Metrics Classification Modifiers
stdcell, macros, clock:<name>, iteration:<num>, etc.

Fig. 1. METRICS2.1 organization.

The following paragraphs give additional details of these aspects of METRICS2.1.

**Stages and snapshots.** METRICS2.1 is organized as a hierarchical JSON object as shown in Figure 1. The top level of the JSON object is the stage or snapshot. A stage is a predefined stage of the design flow. The current stages in METRICS2.1 are `run`, `init`, `synth`, `floorplan`, `globalplace`, `placeopt`, `detailedplace`, `cts`, `globalroute`, `detailedroute` and `finish`.[2] A snapshot can be any user-defined stage with a unique name to capture the metrics at any point during the flow. For example, a user experimenting with two different optimization recipes can create two snapshots (say, `opt_strategy1` and `opt_strategy2`) to capture the same metrics after each recipe for comparison purposes.[3]

**Categories.** Inside each stage or snapshot are individual metrics organized by metrics category. Current metrics categories are `flow` to represent all the flow related metrics; `design` to represent all the metrics related to the design data, including the physical PPA metrics; `timing` to represent all the timing PPA metrics; `clock` to represent all the primary and derived clocks and their values; `route` to represent all routing-related metrics; and `power` to represent all the power PPA metrics. These metric categories are predefined in METRICS2.1; new categories will be added based on future needs and user inputs.[4]

**Names.** Within each category are predefined metrics organized by metric name, along with optional metric name modifier and metric classification modifiers. A set of existing metric names is predefined in the METRICS2.1 documentation, and new names will be added in future revisions based on user inputs.

**Name Modifiers.** A metric name can have an optional predefined name modifier to uniquely define the metric and its

[2]We use specific fonts in the context of the hierarchical naming convention. METRICS2.1 stages, categories, metric names, name modifiers and classification modifiers are shown in `Courier`. Fully-expanded metric names are shown as Roman font in double quotes.

[3]We note that METRICS2.1 allows all metrics to be present at any stage or snapshot, but certain metrics will only make sense at certain flow junctures. For example, route metrics will be sensible post-placement. It is the user’s responsibility to configure which metrics are extracted and reported at which stage, based on the application.

[4]A user-specified delimiter string can be optionally used between the stage or snapshot name and the metrics category, to “flatten” the metrics for the design flow. This can be convenient for particular analysis applications or for ease of mapping to existing metrics data collections.

TABLE I
SAMPLE TIMING AND POWER METRICS

| Metric | Description |
|---|---|
| **timing**__setup__wns | Setup WNS in the design across all clocks. |
| **timing**__setup__wns__clock :clk_a | Setup WNS for clock "clk_a". |
| **timing**__setup__wns__clock :clk_a__path_group:in_reg | Setup WNS for clock "clk_a" for all input to register paths. |
| **timing**__setup__wns__analysis_view :slow | Setup WNS across all clocks for the analysis view "slow". |
| **power**__total | Total Power consumption. |
| **power**__leakage | Total leakage power. |
| **power**__leakage__clock | Total leakage power on the clock network. |

unit. For example, in the design category we have a metric name `instance` and a name modifier `count` to specify the instance count; here, the implied unit would be an integer. Similarly, in the timing category we have a metric name `setup` and a name modifier `wns`, to specify the setup worst negative slack in the design. Many metric names will not require a name modifier: an example is the metric name `switching` in the power category, which specifies the switching power consumption of the design.

**Classification Modifiers.** Optional metric name classification modifiers provide more specific information about a given metric. These can be either type classification modifiers or structure classification modifiers. Type classification modifiers further subdivide the metric into specific subtypes to show a distribution of the metric across the subtypes. An example would be to show the breakdown of the number of instances in the design by stdcells and macros and further break down the number of stdcells into sequential or combinational. Structure classification modifiers, on the other hand, provide information about a specific view of the design. An example would be to provide the information for a specific *analysis view* or a specific *clock domain* for a timing metric such as "timing__setup__wns" or a power metric such as or "power__internal".

Table I shows example usage of metric names, metric name modifiers, and metric classification modifiers for `timing` and `power`. "timing__setup__wns" is in the `timing` category and the metric name is `setup`. `wns` is the name modifier that specifies setup worst negative slack. As there are no other modifiers, this metric is across all clocks in the design. "timing__setup__wns__clock:clk_a" specifies the setup WNS for clock "clk_a". Similarly, we can add more modifiers to specify path groups and analysis views.

The OpenROAD tool [35], an open-source RTL-to-GDS EDA system that integrates a variety of academic tools, supports the METRICS2.1 metric naming standard. The OpenROAD project uses these metrics for internal continuous integration processes, QoR tracking, and management of pull requests in its GitHub repositories. As shown in Figure 2, METRICS2.1 integrates metrics from logs and reports obtained from OpenROAD flow runs. We also use METRICS2.1 to openly share multiple designs of experiments, each containing thousands of RTL-to-GDS metrics datasets along with config files that enable reproducibility, via the IEEE CEDA DATC's RDF repository on GitHub [36]. We also share sample data analyses and ML applications in the form of Jupyter notebooks as a guide to working with large-scale metrics data.

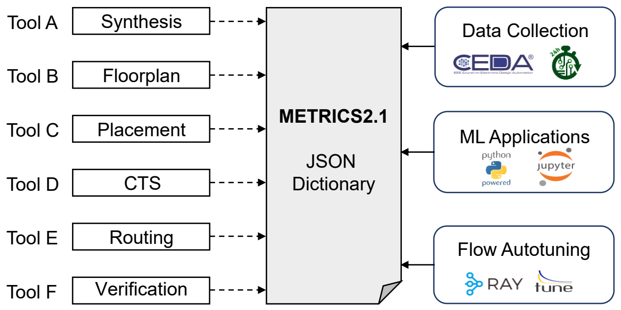


Fig. 2. Overview of METRICS2.1 infrastructure.

## III. USING METRICS2.1 DATA FOR VISUALIZATION AND ML APPLICATIONS

METRICS2.1 supports analysis of how flow parameter settings affect QoR outcomes, as well as the building of machine learning applications to predict tool and flow outcomes. A typical EDA design flow invokes multiple engines, with each engine (e.g., global placement) having multiple parameters to guide the heuristic optimization that it performs. The parameter settings for a given engine will affect not only the results produced by that engine, but also the results of the entire flow. METRICS2.1 provides consistent reporting of metrics across flow stages, enabling collection of OpenROAD run metrics from large-scale designs of experiments that vary tool and flow parameters in a controlled manner. Moreover, because METRICS2.1 captures all parameter settings and commit versions used in a given run, collected data is inherently reproducible. The collected data serves many purposes, e.g., (i) showing evolution of PPA metrics throughout the flow; (ii) enabling recovery of parameter settings that led to particular outcomes; and (iii) giving insights into trends and interrelationships between values of multiple parameters. Typical experiment types and purposes include (i) running one design on one platform to study the impact of parameter settings for that design on that platform; (ii) running multiple designs on the same platform to model and predict outcomes for unseen designs on that platform; and (iii) running multiple designs on multiple platforms to build predictive models.

We regularly publish complete METRICS2.1 data from multiple full experiments in the IEEE CEDA DATC's "Metrics4ML" GitHub repository [36]. Examples of experiments to date include studies of how layer resource reductions in global routing interact with core utilization and/or placement density settings. The repository contains all information needed to reproduce the results (e.g., commit hashes for the OpenROAD app and flow-scripts, and config file settings for running the OpenROAD flow), collected metrics data, and sample Jupyter notebooks that operate on the data. We next outline a sample application that is available in the Metrics4ML GitHub repository.

```
set_global_routing_layer_adjustment met5 0.5
set_global_routing_layer_adjustment met4 0.4
set_global_routing_layer_adjustment met3 0.5
set_global_routing_layer_adjustment met2 0.4
set_global_routing_layer_adjustment met1 0.4

set_routing_layers -signal $MIN_LAYER-$MAX_LAYER
```

Fig. 3. OpenROAD Tcl script to set *layer_adjust* values.

**Layer Resource Adjustment with Core Utilization Sweep.** The OpenROAD flow provides parameters to artificially adjust the track resource for individual routing layers during global routing. These **layer resource adjustments** strongly affect the viability of detailed routing "route guides" that are produced by the global router. While the global router accurately comprehends blockages and track resources on each routing layer, the *layer_adjust* parameter provides further derating of the per-layer track resource. Conceptually, too large a layer resource reduction makes the global router overly pessimistic, and causes inability to resolve global routing congestion (i.e., the global router will fail, giving a "false negative" for the quality of the upstream placement solution). On the other hand, too small a layer resource reduction makes the global router overly optimistic, and its route guides may not be viable in detailed routing. Figure 3 shows an example of setting *layer_adjust* values in the OpenROAD flow scripts.

Another important flow parameter is **core utilization**. Since core utilization is determined by the design core area and design size, it can be varied by changing the design core area for a given design. Lower core utilization implies higher routing capacity for a given design. However, the total wirelength can increase.

We now study the open-source RISC-V *ibex_core* (IBEX) [34] on the SKY130HD platform. This platform has five routing layers, met1 through met5. We vary the *layer_adjust value* of met2, met3 and met4 from 0.1 to 0.2 in steps of 0.02 while keeping the *layer_adjust* value of met5 at 0.5. We also vary the core utilization of the design from 0.2 to 0.4 in steps of 0.02. This results in a total of 6710 runs, of which 6076 are successful and 634 fail with one or more DRC errors in detailed routing. The Jupyter notebook that reads and visualizes the entire dataset can be downloaded from the "Metrics4ML" GitHub repository [36].

As shown in Figure 4, the wirelength generally decreases with increased core utilization until it reaches a sweet spot, beyond which the wirelength starts to increase as the detailed router makes more detours to avoid DRC errors. The sweet spot for this design and technology platform is at around 37% core utilization. We observe in Figure 5 that the detailed router's runtime is fairly constant up to the sweet spot of core utilization; there is very little runtime variance across all combinations of *layer_adjust* values. However, beyond the sweet spot we see a wider variance of the runtime across *layer_adjust* settings, and an overall increase in runtimes. Almost all of the runtime increase is in the detailed router, which performs significantly more ripup and reroute iterations to achieve DRC-correct routing.

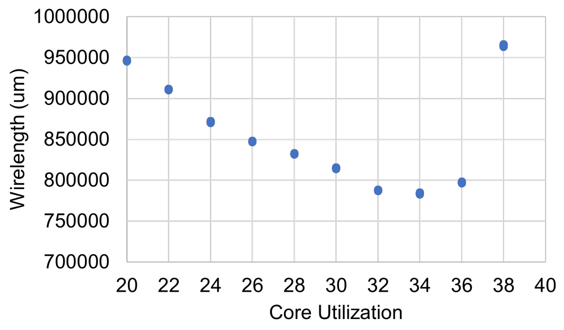


Fig. 4. Wirelength vs. core utilization.

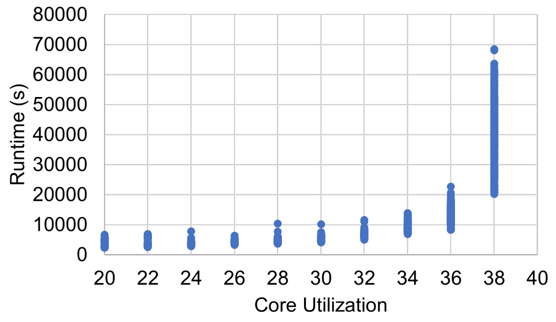


Fig. 5. Detailed router runtime vs. core utilization.

Not surprisingly, the number of DRC errors is 0 until we approach the sweet spot of core utilization, and DRC errors then increase with larger *layer_adjust* values. The Jupyter notebook in GitHub contains implementations of logistic regression modeling to predict runtime, wirelength and doomed runs for new *layer_adjust* and core utilization settings. The models achieve close to 100% accuracy.

## IV. Automatic RTL-to-GDS Flow Tuning

In this section, we describe *AutoTuner*, an automatic RTL-to-GDS flow tuner that leverages the METRICS2.1 infrastructure. As is well-known, the RTL-to-GDS flow is comprised of a series of NP-hard problem formulations, each of which is addressed by complex heuristics (i.e., tools at each flow stage) that have unknown suboptimality gaps. In practice, obtaining good QoR depends on tool parameter settings that are based on an expert designer's experience and knowledge, as well as many sequential iterations. This motivates the use of autotuning, which considers the entire RTL-to-GDS flow as a black-box optimization whose outcome depends on flow/tool parameters.

### A. Related Work

Hyperparameter optimization is the core of any autotuning framework. We review several popular search algorithms for hyperparameter optimizations, as well as their application to EDA flow tuning.

*1) Hyperparameter Search Algorithms:* In hyperparameter optimization, search algorithms look for the best hyperparameter set that minimizes a predefined loss function over a given dataset. Representative search algorithms are as follows.

**Random/grid search.** Random search iteratively picks a parameter configuration randomly from a given parameter space and creates a new trial to evaluate the configuration. Grid search sweeps the entire hyperparameter space. Random search and grid search are model-free approaches that allow fully parallel computation for exploration, but cannot perform exploitation.

**Gradient-based optimization.** Gradient-based algorithms such as gradient descent are used to find optimal values for differentiable functions. In particular, libraries such as *XGBoost* [10] and *LightGBM* [20] have hyperparameter tuning capabilities. However, gradient-based optimization requires more evaluations as the dimension of the objective function increases; a derivative-free approach is more appropriate when the evaluation of the objective function is computationally expensive [3].

**Evolutionary optimization.** Evolutionary optimization is a type of derivative-free optimization inspired by analogies to biological evolution. Well-known examples include genetic algorithms (GA) and particle swarm optimization (PSO). A variety of techniques (e.g., mutation and crossover in GA; inertia and acceleration in PSO) are used to escape from local optima and to improve convergence rates. *Nevergrad* [37] from Facebook provides several algorithms specialized to the tuning of mixed discrete and continuous parameters. Google's *population based training (PBT)* [15] performs mutation at regular intervals, while evaluating multiple steps for a single trial. In addition, *PBT* promotes efficient exploitation by periodically sharing parameters from a high-performance worker to a low-performance worker in a parallel system.

**Bayesian optimization.** Bayesian optimization is a derivative-free optimization that enables fast convergence for noisy and expensive-to-evaluate objective functions. Assuming initially a random function, the Bayesian algorithm creates a prior distribution based on its current belief regarding functional behavior of the objective function. With each successive trial, the prior distribution is updated to a posterior distribution based on an updated behavior belief. The Bayesian algorithm iterates the above process to approximate the objective function to find the best hyperparameter set. The model used to approximate the objective function is called a *surrogate model*; Gaussian processes and Tree Parzen Estimator (TPE) [6] are widely used. Representative Bayesian optimizers include *HyperOpt* [7], *Optuna* [4], *BoTorch* [8], and *Ax* [31].

**Bandit optimization.** Bandit optimization for finite selection sets is not strictly Bayesian optimization. However, aside from targeting discrete (Bandit) as opposed to continuous (Bayesian) parameters and sampling methods, its exploration-exploitation tradeoff tendency is very similar to that of Bayesian optimization. The Bandit approach is optimized for the multi-armed bandit (MAB) problem [19] through, e.g., Thompson sampling [24].

*2) RTL-to-GDS Flow Autotuning:* Over the past several years, a number of researchers have developed autotuning methods in the RTL-to-GDS flow domain. Xu et al. [27] propose an MAB-based autotuner for QoR improvement in academic and commercial FPGA compilation flows, using the open-source OpenTuner [2] autotuning framework. Yu et al. [28] propose a convolutional neural network (CNN) model to solve a design-specific logic synthesis tuning problem as a multi-class classification problem. Hosny et al. [14] present an open-source deep reinforcement learning framework for logic synthesis by iteratively choosing primitive transformations with the highest expected reward. Kwon et al. [21] propose a learning-based parameter recommendation system that trains a post-placement QoR prediction model using archived design data obtained via iterative tuning, along with online recommendation. Ustun et al. [25] propose an FPGA autotuning system with multi-stage QoR information and online learning. Agnesina et al. [1] propose a deep reinforcement learning framework to optimize placement parameters. Xie et al. [26] use feature importance sampling and tree-based parameter tuning to find a best flow parameter configuration. Recently, Ziegler et al. [29] have provided an overview of industrial flow tuning using online and offline autotuning approaches in the RTL-to-GDS domain. In general, previous frameworks are closed-source, requiring substantial pre-archived data [21], focusing on one particular flow stage [1], [14], [28], or having a limited search space with fragmented discrete parameters [26]. This motivates development of an open-source autotuning framework for academic and commercial RTL-to-GDS flows, based on the open METRICS2.1 names and format. Such infrastructure can strengthen connections between academic research works on various flow stages, while also saving redundant implementation efforts.

## B. *AutoTuner Framework*

We now describe *AutoTuner*, which is shown in Figure 6. Given a set of parameters, AutoTuner iteratively tunes the parameters to find a parameter combination that best optimizes a prescribed objective, or *score*, function. Users can define the objective function in terms of metrics available in METRICS2.1. AutoTuner supports multiple search algorithms, and users can specify a particular algorithm to be used. The framework is built upon Ray/Tune [22], [23] to enable parallelized flow parameter tuning. AutoTuner for OpenROAD is open-sourced and shared in the IEEE CEDA DATC's organization GitHub [30].

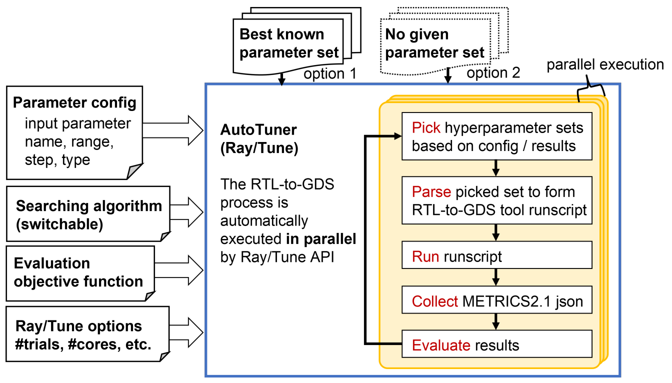
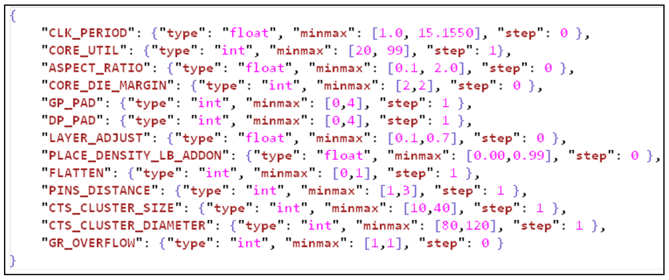

Best known parameter set
No given parameter set
option 1
option 2
parallel execution
Parameter config
input parameter name, range, step, type
Searching algorithm (switchable)
Evaluation objective function
Ray/Tune options #trials, #cores, etc.
AutoTuner (Ray/Tune)
The RTL-to-GDS process is automatically executed in parallel by Ray/Tune API
Pick hyperparameter sets based on config / results
Parse picked set to form RTL-to-GDS tool runscript
Run runscript
Collect METRICS2.1 json
Evaluate results

Fig. 6. Overview of the AutoTuner framework.



```
{
    "CLK_PERIOD": {"type": "float", "minmax": [1.0, 15.1550], "step": 0 },
    "CORE_UTIL": {"type": "int", "minmax": [20, 99], "step": 1},
    "ASPECT_RATIO": {"type": "float", "minmax": [0.1, 2.0], "step": 0 },
    "CORE_DIE_MARGIN": {"type": "int", "minmax": [2,2], "step": 0 },
    "GP_PAD": {"type": "int", "minmax": [0,4], "step": 1 },
    "DP_PAD": {"type": "int", "minmax": [0,4], "step": 1 },
    "LAYER_ADJUST": {"type": "float", "minmax": [0.1,0.7], "step": 0 },
    "PLACE_DENSITY_LB_ADDON": {"type": "float", "minmax": [0.00,0.99], "step": 0 },
    "FLATTEN": {"type": "int", "minmax": [0,1], "step": 1 },
    "PINS_DISTANCE": {"type": "int", "minmax": [1,3], "step": 1 },
    "CTS_CLUSTER_SIZE": {"type": "int", "minmax": [10,40], "step": 1 },
    "CTS_CLUSTER_DIAMETER": {"type": "int", "minmax": [80,120], "step": 1 },
    "GR_OVERFLOW": {"type": "int", "minmax": [1,1], "step": 0 }
}
```

Fig. 7. Sample input config file. In the JSON dictionary format, each key is the name of the parameter and contains data type, min to max range, and step size.

*1) Parameter Configurations:* AutoTuner takes as input a JSON configuration file that defines a space of tool/flow parameters. Figure 7 shows an example. Each parameter is defined by its name, type, range (min, max), and step size within the range. If the step size is 0, an integer parameter is considered to be a fixed constant: the min and max bounds of the range take the same value. For a float parameter, a step size of 0 defines a continuous range.

*2) Objective Function with Tool Noise Consideration:* In general, final metrics of a VLSI design reflect multiple PPA goals, namely, power, performance, and area. In our work, we formulate a single score evaluation function to capture design solution quality with respect to the three PPA metrics. While some search algorithms available via the Ray/Tune API support multi-objective score functions, we integrate the three PPA metrics into a single objective function to allow application of a wider variety of search algorithms in our studies.

Our score evaluation function is defined as

$$\text{Score} = \left(\text{PPA}_{\text{imp}}^{\text{UB}} - \text{PPA}_{\text{imp}}\right) \cdot (1 + \alpha \cdot N_{\text{DRV}}) \qquad (1)$$

where $\text{PPA}_{\text{imp}}^{\text{UB}}$ is the upper bound of PPA improvement, $\text{PPA}_{\text{imp}}$ is the PPA improvement of the current trial, $\alpha$ is a user-specified factor, and $N_{\text{DRV}}$ is the number of DRC violations. The term $\alpha N_{\text{DRV}}$ is used to penalize trials with DRC violations. With this score evaluation function, the best parameter setting is obtained when the score function is minimized.

We use total power, effective clock period, and total cell area as the PPA metrics. (As noted above, the PPA metrics used for flow parameter tuning can be easily changed thanks to the underlying METRICS2.1 framework.) PPA improvement is measured against a reference place-and-route result obtained with manually-tuned parameters; power, effective clock period, and cell area of the reference result are respectively denoted by $P_{\text{ref}}$, $\text{effCP}_{\text{ref}}$, and $A_{\text{ref}}$. Given a trial result with a new parameter configuration, the improvements of power ($\text{Power}_{\text{imp}}$), performance ($\text{Perf}_{\text{imp}}$), and area ($\text{Area}_{\text{imp}}$) are respectively given by

$$\text{Power}_{\text{imp}} = \frac{P_{\text{ref}} - p}{P_{\text{ref}}},\ \text{Perf}_{\text{imp}} = \frac{\text{effCP}_{\text{ref}} - \text{effCP}}{\text{effCP}_{\text{ref}}},\ \text{Area}_{\text{imp}} = \frac{A_{\text{ref}} - a}{A_{\text{ref}}} \qquad (2)$$

where $p$, effCP, and $a$ are total power, effective clock period, and cell area of the current trial. The PPA improvement $\text{PPA}_{\text{imp}}$ is then defined as a weighted sum of the improvements:

$$\text{PPA}_{\text{imp}} = C_P \cdot \text{Power}_{\text{imp}} + C_D \cdot \text{Perf}_{\text{imp}} + C_A \cdot \text{Area}_{\text{imp}} \qquad (3)$$

where $C_P$, $C_D$ and $C_A$ are weights for power, performance and area, respectively. While the suboptimality gap for RTL-to-GDS optimization is unknown, an abstract upper bound on PPA improvement corresponds to power, effective clock period, and area all becoming zero. I.e., $\text{PPA}_{\text{imp}}^{\text{UB}}$ is $C_P + C_D + C_A$.

Inherent EDA tool noise can introduce significant variation of final QoR, even with isomorphic inputs (e.g., identical gate-level netlists that differ only in instance, net, and cell master names [17]). Chan et al. [9] report that with modern EDA tools, noise effects can still cause more than 10% variation in routed wirelength. In our work, we consider tool noise effects on the *exploitation* process in AutoTuner. Specifically, we modify Equation (1) to mimic an ML training epoch:

$$\text{Score} = \left(\text{PPA}_{\text{imp}}^{\text{UB}} - \text{PPA}_{\text{imp}}\right) \cdot \left(\frac{s_{\max}}{s} + \alpha \cdot N_{\text{DRV}}\right) \qquad (4)$$

where $s_{\max}$ is a user-defined maximum step number, and $s$ is the current step. At each step, we add random Gaussian noise to obtained power and performance metrics ($p$ and effCP in Equation (2) in the objective function evaluation. The noise perturbations are sampled from a distribution with zero mean and 3-sigma set to a target percentage of the metric value. (Because area is determined and fixed in the floorplan according to target core utilization, we do not consider noise effects of floorplan area.) In this way, metric noise becomes a proxy for tool noise.

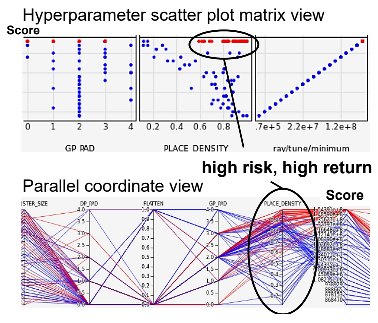
## C. Supported Search Algorithms

The AutoTuner framework can support various search algorithms. In our current implementation, we select competitive search algorithms listed in Table II. Each algorithm has its own pros and cons depending on (i) difficulty of the target black box problem, (ii) number of target hyperparameters, (iii) type of hyperparameters (continuous, discrete, mixed), (iv) level of parallelism (#concurrent jobs), and (v) scheduling of parallelization (synchronous / asynchronous).

TABLE II
SEARCH ALGORITHMS SUPPORTED IN AUTOTUNER

| **Types** | **Name** |
|---|---|
| Random / grid search | Random / grid search |
| Population Based Training | PBT [15] |
| Tree Parzen Estimator (TPE) | HyperOpt [7] |
| Bayesian + Multi-Armed Bandit | AxSearch [8], [31] |
| TPE+CMA-ES | Optuna [4] |
| Evolutionary Algorithm | Nevergrad [37] |

To select appropriate search algorithms, we consider relevant aspects of the RTL-to-GDS flow domain. Running the RTL-to-GDS flow is expensive, and each trial consumes substantial runtime. Furthermore, the hyperparameter space contains many combinations that lead to failed runs; for example, run failure can be common when parameters such as utilization or target clock period take on extreme values in their given ranges. On the other hand, restricting the parameter space to avoid infeasibility or failure will miss high-quality parameter combinations, as seen with the example autotuning run shown in Figure 8. Moreover, the RTL-to-GDS flow has a mix of continuous and discrete hyperparameters. This is an important practical consideration because some Bayesian optimization algorithms do not support discrete parameters.



| Parameters | Human | AutoTuner |
|---|---|---|
| Flatten | 1 | 0 |
| Global placement padding | 4 | 2 |
| Detailed placement padding | 2 | 0 |
| IO pin distance (um) | 2 | 2 |
| CTS clustering size (um) | 30 | 37 |
| CTS clustering diameter (um) | 100 | 95 |
| Layer resource adjustment | 0.5 | 0.42 |
| Target GP density | **0.6** | **0.99** |

Fig. 8. Autotuning results for the IBEX design in SKY130HD. Despite many failed trials (red dots) at high target local placement density, the best result is achieved with placement density = 0.99, where the maximum possible value is 1.00.

## V. AutoTuner Case Studies

We now describe case studies using our flow autotuning framework. The AutoTuner framework is written in Python 3.8. Experiments for the case studies are performed on a Google Cloud Platform `m1-ultramem-160` instance equipped with 160 2.2GHz CPUs and 3.8TB RAM. We use the OpenROAD RTL-to-GDS tool chain with ASAP7 and SKY130HD. Three public designs, *aes_cipher* (AES) [32], *jpeg_encoder* (JPEG) [33], and IBEX, are used as testcases. We set $\alpha$ and $s_{max}$ in the objective function (Equation (4)) as 0.1 and 100, respectively. In all experiments, asynchronous parallelization is used to achieve more aggressive explorations. Except for results shown in Section V-B, all case studies are parallelized to 40 concurrent OpenROAD jobs, with each job being assigned to 4 cores. The tunable tool parameters used for the case studies are listed in Table III.[5]

### A. Comparison of Search Algorithms

We make an initial comparison of the search algorithms listed in Table II, using our AutoTuner framework. Each search algorithm has a different tradeoff of exploration versus exploitation, and a given trial can have highly variable runtime. For example, early-stage failure with infeasible parameters will result in small runtime, while tight inputs can lead to long detailed routing runtime as shown in Figure 5. Therefore, we compare the algorithms based on a fixed walltime constraint instead of total number of trials. Since the *PBT* algorithm generates mutations at regular step intervals, we set the walltime constraint as 63966 seconds through 250 workers, 100 steps, and 25-step interval for mutation in the *PBT* algorithm; with this walltime, *PBT* executes 1000 RTL-to-GDS flow trials.

[5]The lower bound of CLOCK_PERIOD, $T_{min}$, is set to 1ns and 100ps for SKY130HD and ASAP7, respectively; the upper bound, $T_{max}$, is set to a user-specified value for each design.

TABLE III
Tunable Tool and Design Parameters

| Parameters | Description | Type | Range |
|---|---|---|---|
| CLOCK_PERIOD | Target clock period (ns) | float | $[T_{min}, T_{max}]$ |
| CORE_UTIL | Target core utilization (%) | int | [20, 99] |
| ASPECT_RATIO | Floorplan aspect ratio | float | [0.1, 2.0] |
| GP_PAD | Cell padding for global placement (site) | int | [0, 4] |
| DP_PAD | Cell padding for detailed placement (site) | int | [0, 4] |
| LAYER_ADJUST | Layer resource adjustment for global routing (%) | float | [0.1, 0.7] |
| PLACE_DENSITY _LB_ADDON | Additional lower bound increase of the target local global placement density (%) | float | [0.00, 0.99] |
| FLATTEN | Design hierarchy flattening | int | [0, 1] |
| PINS_DISTANCE | Minimum IO pin distance (#tracks) | int | [1, 3] |
| CTS_CLUSTER _SIZE | Target CTS sink cluster size | int | [10, 40] |
| CTS_CLUSTER _DIAMETER | Target CTS sink cluster diameter (um) | int | [80, 120] |

Table IV shows the comparison between search algorithms. The PPA weights $C_P$, $C_D$, and $C_A$ in Equation (4) are 100, 10000 and 100, respectively. The columns labeled "Imp." give improvement over the reference runs; the third column lists the best effective clock period obtained via tuning. For *Nevergrad*, we use the *PortfolioDiscreteOnePlusOne* engine, following the authors' guidance for better performance with mixed continuous and discrete parameters. *Ax* does not maintain the set number of current jobs despite asynchronous parallelization settings, so the total number of trials is relatively small. *Nevergrad* spends numerous trials on exploration of loose hyperparameters or infeasible sets. *HyperOpt* and *Optuna* show better (i.e., smaller) final scores, but we note that the appropriate search algorithm may vary depending on the design difficulty. Based on this initial study, we conduct subsequent case studies using *HyperOpt*.

TABLE IV
Autotuning Algorithm Comparison with Timeout Constraints

| Algorithm | #Trials | effCP (ns) | Imp. | Util (%) | Imp. | Power (W) | Imp. | Score |
|---|---|---|---|---|---|---|---|---|
| **Reference** | | 17.78 | 0% | 20 | 0% | 0.0523 | 0% | 1020000 |
| **PBT** | 1000 | 15.16 | 15% | 37 | 46% | 0.0143 | 73% | 863611 |
| **HyperOpt** | 742 | 14.94 | 16% | 40 | 50% | 0.0286 | 45% | 853503 |
| **Ax** | 478 | 16.04 | 10% | 26 | 23% | 0.0172 | 67% | 914902 |
| **Optuna** | 1493 | 14.87 | 16% | 42 | 52% | 0.0278 | 47% | 849094 |
| **Nevergrad** | 2925 | 15.23 | 14% | 18 | -11% | 0.0194 | 63% | 870843 |

### B. Exploration versus Exploitation

Hyperparameter optimization algorithms can apply asynchronous parallelization. However, for a given total number of trials, high parallelization can result in performance degradation by reducing the number of iterations for exploitation. For example, using $N$ concurrent jobs for $N$ trials devolves to random multistart search. We study walltime reduction versus score improvement from autotuning, as the number of concurrent jobs is changed. In this study, we use the SKY130HD IBEX testcase; we also use post-route wirelength with DRC penalty as the evaluation (i.e., score) function to reduce the incidence of infeasible parameter combinations. Autotuning uses eight parameters corresponding to those shown in Table III, except for the footprint and timing constraints parameters (CLOCK_PERIOD, CORE_UTIL and ASPECT_RATIO).

As shown in Figure 9, as the degree of parallelization (#Concurrent runs) increases, the total walltime (left $y$-axis) decreases by up to 79%. However, the wirelength objective (right $y$-axis) degrades somewhat with increased parallelization, even though all autotuning results show substantial improvement over the human-tuned reference. We observe that asynchronous parallelization yields walltime that scales with $\frac{1}{\#concurrent\_jobs}$, while the performance degradation of the search algorithm scales with $\frac{\#concurrent\_jobs}{\#trials}$. Therefore, if the number of trials is large enough, active parallelization significantly reduces the latency of autotuning without compromising quality of results.

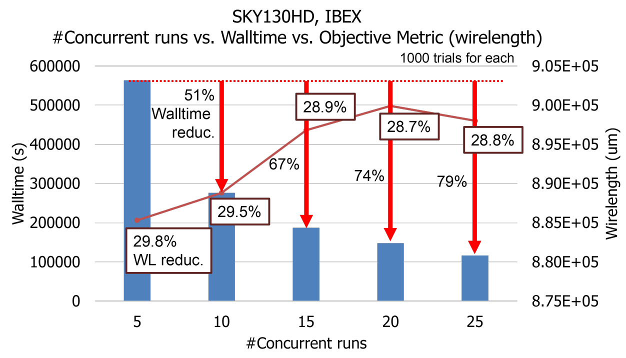


Fig. 9. Walltime (left $y$-axis bar) versus objective metric (wirelength, right $y$-axis line) with different numbers of concurrent jobs (parallelization).

### C. Validation of Target Objective Tuning

In IC implementation, the three component metrics of PPA quality typically exhibit a Pareto tradeoff across high-quality solutions. We study how AutoTuner can be steered to emphasize particular PPA component metrics by adjusting the three coefficients in our score function, using multiple platforms and designs. Table V describes the results for AES, JPEG, IBEX designs in the *ASAP7, SKY130HD* platforms with four coefficient settings: **{power, performance, area, uniform}**. As shown in Table V, our score evaluation function can effectively steer the result according to which metric is emphasized. In the SKY130HD and ASAP7 platforms, respectively, we achieve an average of (41%, 29%) total power consumption reduction with the **power** setting, (21%, 18%) effective clock period improvement with the **performance** setting, and (68%, 50%) area reduction with the **area** setting.

## VI. Conclusion

In this paper, we have proposed METRICS2.1, a standardized format for design tool and flow metrics data that provides a robust structure for large-scale metrics archives. METRICS2.1 is supported by a new initiative of the IEEE CEDA DATC, and is now a part of the DATC RDF. Together, METRICS2.1, RDF and the OpenROAD flow fill a long-standing gap in the enablement of EDA point tool research. Notably, developers of an improved engine (that has well-defined inputs and outputs) can immediately assess impacts on PPA and other metrics not only for the engine (e.g., by substitution into a given flow step), but for the entire RTL-to-GDS flow as well. Our infrastructure also provides support for academic contests that advance point optimizations (e.g., CTSOpt, useful skew, routability- and IR-driven PDN, macro placement, etc.) with industry-standard data and formats, and an overall flow context.

We have also described *AutoTuner*, an open-source design flow autotuning framework. AutoTuner implements "no-human-in-loop" parameter tuning for commercial and academic RTL-to-GDS flows, and is available at the the DATC's organization GitHub [30]. Using the AutoTuner framework, we have presented assessments of various search algorithms, steering of PPA enhancements in ASAP7 and SKY130HD technologies, and tradeoffs between exploration and exploitation seen with parallelization. Our ongoing work seeks to optimize computing resource usage as well as PPA outcomes within prescribed schedule and compute constraints, using remote distributed computing resources and additional strategies such as early-stage termination of unpromising trials.

TABLE V
PPA objective tuning with four different settings. Each setting uses 1000 trials. All post-routing results are DRC-clean.

| **Platform** | **Design** | **Setting** | **effCP** (ns) | **Imp.** | **Util** (%) | **Imp.** | **Power** (W) | **Imp.** |
|---|---|---|---|---|---|---|---|---|
| | | ref. | 4.554 | 0% | 19 | 0% | 0.0838 | 0% |
| | | power | 4.373 | 4% | 23 | 17% | **0.0569** | **32%** |
| | AES | perf. | **3.491** | **23%** | 22 | 14% | 0.0942 | -12% |
| | | area | 3.763 | 17% | **42** | **55%** | 0.0558 | 33% |
| | | uniform | 3.380 | 26% | 35 | 46% | 0.0612 | 27% |
| | | ref. | 9.277 | 0% | 20 | 0% | 0.0905 | 0% |
| | | power | 7.781 | 16% | 37 | 46% | **0.0711** | **21%** |
| SKY130HD | JPEG | perf. | **7.259** | **22%** | 31 | 35% | 0.1350 | -49% |
| | | area | 12.186 | -31% | **89** | **78%** | 0.1010 | -12% |
| | | uniform | 7.880 | 15% | 84 | 76% | 0.0800 | 12% |
| | | ref. | 17.775 | 0% | 19 | 0% | 0.0461 | 0% |
| | | power | 16.138 | 9% | 39 | 51% | **0.0133** | **71%** |
| | IBEX | perf. | **14.656** | **18%** | 33 | 42% | 0.0162 | 65% |
| | | area | 18.552 | -4% | **64** | **70%** | 0.0408 | 11% |
| | | uniform | 15.502 | 13% | 54 | 65% | 0.0135 | 71% |
| **Platform** | **Design** | **Setting** | **effCP** (ps) | **Imp.** | **Util** (%) | **Imp.** | **Power** (W) | **Imp.** |
| | | ref. | 497.630 | 0% | 29 | 0% | 0.0170 | 0% |
| | | power | 466.449 | 6% | 36 | 19% | **0.0089** | **48%** |
| | AES | perf. | **415.522** | **16%** | 41 | 29% | 0.0241 | -42% |
| | | area | 452.110 | 9% | **70** | **59%** | 0.0172 | -1% |
| | | uniform | 458.444 | 8% | 70 | 59% | 0.0164 | 4% |
| | | ref. | 1039.200 | 0% | 30 | 0% | 0.0348 | 0% |
| | | power | 1023.650 | 1% | 59 | 49% | **0.0343** | **1%** |
| ASAP7 | JPEG | perf. | **823.654** | **21%** | 56 | 46% | 0.0645 | -85% |
| | | area | 925.933 | 11% | **68** | **56%** | 0.0448 | -29% |
| | | uniform | 998.296 | 4% | 68 | 56% | 0.0395 | -14% |
| | | ref. | 1715.330 | 0% | 24 | 0% | 0.0109 | 0% |
| | | power | 1766.009 | -3% | 28 | 14% | **0.0066** | **39%** |
| | IBEX | perf. | **1409.050** | **18%** | 31 | 23% | 0.0113 | -4% |
| | | area | 1433.542 | 16% | **37** | **35%** | 0.0160 | -47% |
| | | uniform | 1485.705 | 13% | 33 | 27% | 0.0086 | 21% |

## Acknowledgments

We thank Colin Holehouse, Tuck-Boon Chan, Jiajia Li, Roger Carpenter, Siddhartha Nath, Mohamed Shalan and the Efabless.com team for inputs toward METRICS2.1 scope and naming. Abdelrahman Hosny, Vitor Bandeira, Matt Liberty, Mehdi Saligane, Wenbo Duan, Indira Iyer, and engine developers have contributed to metrics naming, reporting/messaging, and extraction in OpenROAD. This work is partially supported by DARPA HR0011-18-2-0032 and NSF CCF-2112665.

## References

[1] A. Agnesina, K. Chang and S. K. Lim, "VLSI Placement Parameter Optimization using Deep Reinforcement Learning," *Proc. ICCAD*, 2020, pp. 1–9.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," *Proc. PACT*, 2014, pp. 303–316.

[3] G. I. Diaz, A. Fokoue-Nkoutche, G. Nannicini and H. Samulowitz, "An Effective Algorithm for Hyperparameter Optimization of Neural Networks," *IBM Journal of Research and Development* 61(4/5) (2018), pp. 9:1–9.11.

[4] T. Akiba, S. Sano, T. Yanase, T. Ohta and M. Koyama, "Optuna: A Next-generation Hyperparameter Optimization Framework," *Proc. KDD*, 2019, pp. 2623–2631.

[5] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo and B. Xu, "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project," *Proc. DAC*, 2019, pp. 76:1–76:4.

[6] J, Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for Hyper-Parameter Optimization," *Advances in Neural Information Processing Systems* 24 (2011), pp. 1–9.

[7] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins and D. D. Cox, "Hyperopt: A Python Library for Model Selection and Hyperparameter Optimization," *Computational Science & Discovery* 8(1) (2015), pp. 1–24.

[8] M. Balandat, B. Karrer, D. Jiang, S. Daulton, B. Letham, A. G. Wilson and E. Bakshy, "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization," *Proc. NeurIPS*, 2020, pp. 1–34.

[9] T. B. Chan, A. B. Kahng and M. Woo, "Revisiting Inherent Noise Floors for Interconnect Prediction," *Proc. SLIP*, 2020, pp. 1–7.

[10] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," *Proc. KDD*, 2016, pp. 785–794.

[11] S. Fenstermaker, D. George, A. B. Kahng, S. Mantik and B. Thielges, "METRICS: A System Architecture for Design Process Optimization," *Proc. DAC*, 2000, pp. 705–710.

[12] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro and D. Sculley, "Google Vizier: A Service for Black-Box Optimization," *Proc. KDD*, 2017, pp. 1487–1495.

[13] S. Hashemi, C. T. Ho, A. B. Kahng, H. Y. Liu and S. Reda, "METRICS 2.0: A Machine-Learning Based Optimization System for IC Design," *Workshop on Open-Source EDA Technology*, 2018, pp. 1–4. https://woset-workshop.github.io/PDFs/2018/a21.pdf

[14] A. Hosny, S. Hashemi, M. Shalan and S. Reda, "Drills: Deep Reinforcement Learning for Logic Synthesis," *Proc. ASP-DAC*, 2020, pp. 581–586.

[15] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando and K. Kavukcuoglu, "Population Based Training of Neural Networks," *arXiv preprint 1711.09846*, 2017.

[16] A. B. Kahng and S. Mantik, "A System for Automatic Recording and Prediction of Design Quality Metrics," *Proc. ISQED*, 2001, pp. 81–86.

[17] A. B. Kahng and S. Mantik, "Measurement of Inherent Noise in EDA Tools," *Proc. ISQED*, 2002, pp. 206–211.

[18] A. B. Kahng and T. Spyrou, "The OpenROAD Project: Unleashing Hardware Innovation," *Proc. Government Microcircuit Applications and Critical Technology Conference*, 2021, pp. 1-6.

[19] M. N. Katehakis and A. F. Veinott, Jr., "The Multi-Armed Bandit Problem: Decomposition and Computation," *Mathematics of Operations Research* 12(2) (1987), pp. 262–268.

[20] G. Ke, et al., "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," *Advances in Neural Information Processing Systems* 30 (2017), pp. 3146–3154.

[21] J. Kwon, M. M. Ziegler and L. P. Carloni, "A Learning-based Recommender System for Autotuning Design Flows of Industrial High-performance Processors," *Proc. DAC*, 2018, pp. 1–6.

[22] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training," *arXiv preprint 1807.05118*, 2018.

[23] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," *Proc. OSDI*, 2018, pp. 561–577.

[24] W. R. Thompson, "On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika* 25(3/4) (1933), pp. 285–294.

[25] E. Ustun, S. Xiang, J. Gui, C. Yu and Z. Zhang, "LAMDA: Learning-Assisted Multi-stage Autotuning for FPGA Design Closure," *Proc. FCCM*, 2019, pp. 74–77.

[26] Z. Xie, G.-Q. Fang, Y.-H. Huang, H. Ren, Y. Zhang, B. Khailany, S.-Y. Fang, J. Hu, Y. Chen and E. C. Barboza, "FIST: A Feature-Importance Sampling and Tree-based Method for Automatic Design Flow Parameter Tuning," *Proc. ASP-DAC*, 2020, pp. 19–25.

[27] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo and Z. Zhang, "A Parallel Bandit-based Approach for Autotuning FPGA Compilation," *Proc. FPGA*, 2017, pp. 157–166.

[28] C. Yu, H. Xiao and G. De Micheli, "Developing Synthesis Flows without Human Knowledge," *Proc. DAC*, 2018, pp. 1–6.

[29] M. M. Ziegler, J. Kwon, H.-Y. Liu and L. P. Carloni, "Online and Offline Machine Learning for Industrial Design Flow Tuning," *Proc. ICCAD*, 2021, pp. 1–8.

[30] DATC RDF AutoTuner. https://github.com/ieee-ceda-datc/datc-rdf-flow-tuner.

[31] Ax. https://ax.dev/

[32] AES (Rijndael) IP Core. https://opencores.org/projects/aes_core.

[33] JPEG encoder https://opencores.org/projects/video_systems.

[34] Ibex RISC-V Core. https://github.com/lowRISC/ibex.

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

[36] Metrics4ML. https://github.com/ieee-ceda-datc/datc-rdf-Metrics4ML.

[37] Nevergrad - A gradient-free optimization platform. https://gitHub.com/FacebookResearch/Nevergrad.