IEEE CEDA DATC: Expanding Research Foundations for IC Physical Design and ML-Enabled EDA
Invited Paper

Jinwook Jung
IBM Research
Yorktown Heights, NY, USA
jinwookjung@ibm.com

Andrew B. Kahng
UC San Diego
La Jolla, CA, USA
abk@ucsd.edu

Ravi Varadarajan
UC San Diego
La Jolla, CA, USA
rvaradarajan@ucsd.edu

Zhiang Wang
UC San Diego
La Jolla, CA, USA
zhw033@ucsd.edu

ABSTRACT

This paper describes new elements in the RDF-2022 release of the DATC Robust Design Flow, along with other activities of the IEEE CEDA DATC. The RosettaStone initiated with RDF-2021 has been augmented to include 35 benchmarks and four open-source technologies (ASAP7, NanGate45 and SkyWater130HS/HD), plus timing-sensitive versions created using path-cutting. The Hier-RTLMP macro placer is now part of DATC RDF, enabling macro placement for large modern designs with hundreds of macros. To establish a clear baseline for macro placers, new open-source benchmark suites on open PDKs, with corresponding flows for fully reproducible results, are provided. METRICS2.1 infrastructure in OpenROAD and OpenROAD-flow-scripts now uses native JSON metrics reporting, which is more robust and general than the previous Python script-based method. Calibrations on open enablements have also seen notable updates in the RDF. Finally, we also describe an approach to establishing a generic, cloud-native large-scale design of experiments for ML-enabled EDA. Our paper closes with future research directions related to DATC’s efforts.

CCS CONCEPTS
- Hardware → Physical design (EDA); Methodologies for EDA.

KEYWORDS
VLSI CAD, open source, EDA, machine learning

ACM Reference Format:

1 INTRODUCTION

IEEE CEDA Design Automation Technical Committee (DATC) [21] has developed a public reference design flow, named DATC Robust Design Flow (RDF) [2–4, 10, 11]. The RDF preserves leading research codes in a complete design flow, and serves as a repository of academic point tools. It is also intended to foster flow-scale and cross-stage optimization research, rather than single-stage, “point-tool” optimizations. To this end, RDF also enables measurement of quality of results (QoR) throughout the tool flow, using a standardized metrics format. The first release, named OpenDesign Flow Database, appeared in 2016 and was built upon CAD contest-winning tools. The RDF subsequently evolved both vertically and horizontally to achieve a complete RTL-to-GDS flow with multiple tool options available [2–4, 10, 11]. In the 2020 release [3], RDF brought the integrated OpenROAD app [23] into its inventory, solidifying the RTL-to-GDS implementation flow. The RDF scope and mission were also updated, bringing attention to analysis and verification research; validation of research in a full-flow context; and infrastructure (from obfuscation and anonymization to metrics collection) to support ML-enabled EDA (ML EDA) research. RDF is currently built upon many academic tools, as shown in Table 1.

With RDF as a foundation, the IEEE CEDA DATC is extending its activities beyond flow enablement, with the goal of establishing and expanding research foundations for IC physical design and ML EDA. In the following, we describe three main directions of effort from the past year.

- First, there have been continuous improvements to the existing RDF elements. (i) The RosettaStone format conversion capability, based on the OpenDB data model and in-memory database, has been improved to create more timing-sensitive netlists. (ii) The METRICS2.1 infrastructure in OpenROAD [23] and OpenROAD-flow-scripts [24] has been updated to report metrics in the METRICS2.1 format natively in JSON. (iii) The Calibrations effort [51] has been extended to include additional analysis data to guide academic researchers. This effort compiles datasets for algorithm and machine learning research that aims to improve analysis and verification accuracy, especially in open-source tools. The need for the research community to carefully maintain the quality and accuracy of its open enablements is highlighted with a recent example of RC extraction in the SKY130 technology.

- Second, macro placement is now a separate engine (i.e., no longer embedded within the floorplanning step) in the RDF flow taxonomy. Our efforts have focused on benchmark creation as well as improvements to macro placement capability in RDF. (i) To
drive physical design research with relevant testcases, we incorporate macro-dominated benchmarks that are based on modern open-source designs and open-source PDKs, along with mixed academic-commercial tool flows and corresponding reference results. Here, a recent policy change by Cadence Design Systems [8] enables a larger solution space for flow research, and we provide early examples of what is possible. (ii) We add two new macro placers into RDF-2022: RTL-MP [17], and Hier-RTILMP [16]. The latter works with a multilevel physical hierarchy that is derived from the RTL logical hierarchy, allowing it to handle large IP blocks that have hundreds or even thousands of macros.

- Third, we establish a generic cloud-native enablement for large-scale design of experiments. As is well-known, interest in ML-enabled EDA has been increasing, but generating large amounts of data for ML EDA can require substantial compute resources. We therefore call attention to the wide availability of cloud computing and describe the use of public cloud resources to generate large-scale data for ML EDA research.

The remainder of this paper is organized as follows. Section 2 describes improvements to existing elements of RDF, timing-sensitivity in RosettaStone, deployment of METRICS2.1, and amplification of the Calibrations effort. Section 3 describes efforts related to automatic macro placement. Section 4 describes the cloud-native enablement of large-scale designs of experiments. Section 5 concludes with a summary of current plans and open directions for DATC activity.

2 RECENT IMPROVEMENTS TO RDF

This section reviews three main avenues of improvement made in RDF-2022: timing-sensitivity in RosettaStone, deployment of METRICS2.1, and amplification of the Calibrations effort. Section 3 describes efforts related to automatic macro placement. Section 4 describes the cloud-native enablement of large-scale designs of experiments. Section 5 concludes with a summary of current plans and open directions for DATC activity.

2.1 Timing-Sensitivity of RosettaStone

The DATC RDF includes academic contest benchmarks and contest-winning tools from more than 15 years ago. For some years, these benchmarks and tools were stuck in a "parallel universe": the old tools could not be run on modern designs, and the old benchmarks could not be fed to modern tools. To cope with this, an open-source benchmark conversion platform, RosettaStone, was added in RDF-2021. Figure 1 illustrates the scope of RosettaStone: by leveraging an industry-strength (LEF5.8) data model and in-memory database, RosettaStone can go beyond tool chaining and enable deeper integrations whereby (i) flow stages can consider or co-operate with subsequent stages; or (ii) academic tools can be cross-evaluated on both commercial and academic benchmarks. As elaborated in [13], RosettaStone not only connects past academic tools to the present and future of physical design research, but also enables academic research and contests to be framed in a complete flow context, with canonical evaluations such as post-route timing or number of DRC violations.

A limitation of RosettaStone in RDF-2021 is that it only leverages OpenDB [20], which does not store timing information. In RDF-2022, we add the use of OpenSTA [48] in RosettaStone to avoid generation of unrealistic timing paths during conversion between Bookshelf and LEF/DEF formats. The unrealistic timing paths are typically caused by original contest creators’ remapping of logic gates (e.g., for obfuscation purposes) without consideration of timing constraints. To address this issue, RosettaStone now includes a tunable logic path cutting flow. We call OpenSTA APIs to retrieve the worst timing paths and convert single-output/single-input combinational logic gates into flip-flops, or else insert flip-flops and create associated output nets, so as to not exceed a user-defined maximum path length. In practice, the timing-aware logic cutting flow eliminates the strange timing structures that can exist in old academic benchmarks.

Table 2 shows the results of logic cutting on DAC-2012 benchmarks. PreLC ECP (resp. PostLC ECP) is the original, pre-logic-cutting (resp. post-logic-cutting) effective clock period. PreLC #Stage on WTP (resp. PostLC #Stage on WTP) is the pre-logic-cutting (resp. post-logic-cutting) number of stages on the worst timing path.

![Table 2: Effect of logic cutting on timing sensibility of DAC-2012 benchmarks. PreLC ECP (resp. PostLC ECP) is the original, pre-logic-cutting (resp. post-logic-cutting) effective clock period. PreLC #Stage on WTP (resp. PostLC #Stage on WTP) is the pre-logic-cutting (resp. post-logic-cutting) number of stages on the worst timing path.](image-url)
2.2 METRICS2.1

Both OpenROAD and OpenROAD-flow-scripts (ORFS) have been updated to report metrics in the METRICS2.1 [6, 7, 9] format using native JSON reporting. Previously, the metrics were reported by a Python script which "mined" tool logfiles to extract the relevant metrics information and report it in the METRICS2.1 format. This was error-prone, as the Python script had to be updated frequently to keep track of changes to the logfiles generated by the various engines in OpenROAD.

The change to native JSON-based METRICS2.1 reporting is in the latest public releases of OpenROAD and ORFS. Usage of this reporting is outlined in the steps below.

- Invoke OpenROAD with `-metrics <metric_file_name.json>`. This will print all of the collected metrics into the json file. Some metrics are implicitly collected during the execution of an engine such as global placement, global or detailed routing, etc.; examples include wirelength and DRC errors during each iteration of the detailed router. ORFS also has Tcl commands to explicitly print aggregated metrics such as area, power or timing metrics at any stage in the flow.

- Use the Tcl command `set_metrics_stage "<metrics_stage>"` to set the metrics stage in the flow. This stage is used for all the metrics printed during the OpenROAD call until the stage is changed with a subsequent call to `set_metrics_stage`. The current stages in METRICS2.1 are
  - `set_metrics_stage "floorplan__{}"
  - `set_metrics_stage "globalplace__{}"
  - `set_metrics_stage "placeopt__{}"
  - `set_metrics_stage "detailedplace__{}"
  - `set_metrics_stage "cts__{}"
  - `set_metrics_stage "globalroute__{}"
  - `set_metrics_stage "detailedroute__{}"
  - `set_metrics_stage "finish__{}"

- Use Tcl commands `push_metrics_stage` and `pop_metrics_stage` to apply metrics modifiers to print metrics at different substages as required. For example, ORFS uses the following commands to print metrics pre- and post-timing repair during CTS.
  - `push_metrics_stage "cts__{}__pre_repair"
  - `push_metrics_stage "cts__{}__post_repair"

Figure 2 shows an example of the metrics reported during detailed routing. The detailed router in OpenROAD can print wirelength and number of DRC errors after each iteration, based on a metrics modifier for the `detailedroute__route__wirelength` and `detailedroute__route__drc_errors` metrics. The metrics modifier is `iter:<iteration_count>`.

Figure 3 shows how metrics modifiers change the pre- and post-timing optimization metrics reporting after CTS buffer insertion.

2.3 On the Use of Open Enablements

With continued development and adoption of the RDF and OpenROAD, we observe that open enablements (tools, PDKs, libraries, etc.) are only as good as how they are used. The open-source context allows both tools and enablements to degrade in the absence of strong regression testing, and/or proper checks and balances in the development and release process. A concrete example is seen in the use of the OpenRCX 2.5D RC extraction tool for the open-source SKY130 library [27], with the corresponding open-source SKY130 PDK [27] derived from S8 [28]. The OpenROAD use of OpenRCX for the open-source SKY130HD enablement has shown good correlation for years, as seen in the Cadence Ostrich tool plot of Figure 4. However, OpenLane [25] (commit hash: 7b15116) modifies the lookup tables that guide the
which makes the related ML models much easier to train and un-
ML-enabled macro placement.

(iii) We note implications for research directions in

Hier-RTLMP

for these benchmarks. (ii) A new macro placer, Hier-RTLMP, can

OpenLane. See Figure 6(c).

OpenROA

is the physical synthesis-based SP&R flow using Cadence

Genus iSpatial and Innovus. See Figure 6(b).

Implementation flows enable fully reproducible example macro

placement solutions for each of the testcases and enablements.

These include the following.

• Flow-1 is the logical synthesis-based SP&R flow using Cadence

Genus and Innovus.1 See Figure 6(a).

• Flow-2 is the physical synthesis-based SP&R flow using Cadence

Genus iSpatial and Innovus. See Figure 6(b).

• Flow-3 is the logical synthesis-based SP&R flow using Yosys and

OpenROAD. See Figure 6(c).

All runscripts, including for both macro placement and standard
cell SP&R, are provided for these flows to ensure full transparency
and reproducibility. Here, we highlight the very significant recent
change in policy by Cadence Design Systems [8], which opens up
new synergies between RDF and commercial EDA tools and flows.

The new Cadence policy allows researchers to openly share tool
runscripts; this is a big step toward mitigating the “irreproducibil-
ity of research, by construction” noted in previous RDF updates.

3.2 A Macro Placer for Large-Scale Designs

Macro placement in RDF2019 was performed by

TritonMacroPlace,

which divides the layout region into four quadrants and uses a

1These scripts were written and developed by ABKGroup students at UCSD; however,
the underlying commands and reports are copyrighted by Cadence. We thank Cadence
for granting permission to share our research to help promote and foster the next
generation of innovators.
Hier-RTLMP has been integrated into the latest OpenROAD flow and run on the benchmarks mentioned in Subsection 3.1. Figure 7 shows results from running Hier-RTLMP on a large-scale, complex machine learning accelerator \textit{tabla} 02 with 760 macros, from the open-source project \cite{5}. (The Hier-RTLMP macro placement is the starting point for commercial P&R flow.) Figures 7(a)–(c) show views of the post-routing layout. Figure 7(d) shows the placement of the child clusters of the root (top-level) cluster, along with the corresponding bus planning result. There is one IO cluster containing memories, and eight functional units (PU 0 to PU 7) each of which is an individual cluster containing both macros and standard cells. Standard-cell clusters at the top level contain muxing logic that processes the IOs and interfaces with the eight functional units. The figure shows how the macro placement follows the design’s dataflow, with the IO cluster close to the IOs, and the standard-cell clusters in the middle of the eight functional units. In Figure 7(d), black lines show inter-cluster connections and the dark rectangles show the result of bus planning and pin access region definition along the cluster boundaries. Dark blue rectangles in Figure 7(d) correspond to orange rectangles in Figure 7(b) that highlight pin access regions. Hier-RTLMP creates the bus plan and pin access regions to help ensure global route access into the physical clusters.

### 3.3 ML-Driven Macro Placement

Machine learning approaches have been applied to different stages of physical design, such as standard cell placement \cite{1} and clock tree synthesis \cite{18}. Two basic directions can be seen: (i) the ML-aided approach, which tries to use ML techniques to enhance existing tools or optimizers; and (ii) the ML-driven approach, which tries to use ML techniques to “replace” existing tools or optimizers. Each of these directions is applicable to macro placement.

#### 3.3.1 ML-aided Macro Placement

EDA tools usually have many knobs and parameters, such as the timing-driven and congestion-driven options and effort levels seen with standard-cell P&R tools. Traditionally, tool developers set default values for these knobs and human design engineers tune the knobs further to achieve better PPA. However, the manual tuning is time-consuming and inefficient. Agnesina et al. \cite{1} use a deep reinforcement learning framework to optimize the placement knobs or parameters of a commercial EDA tool. Such a methodology can be also be applied to a macro placer such as Hier-RTLMP. The cost function of Hier-RTLMP is \cite{17}

\[
\text{cost} = \alpha \times \text{Area} + \beta \times \text{WL} + \gamma \times \text{poutline} + \xi \times \text{pBias} + \eta \times \text{Ploutline} + \theta \times \text{pGuidance} + \lambda \times \text{pNotch}
\]

where \text{Area} is the area of the current floorplan, \text{W} is a wirelength estimate (HPWL), \text{poutline} is the penalty for violating the fixed
Hier-RTLMP shows the macro placement generated by weights. Currently, the hyperparameter tuning tool Tune available from [47] and linked documents. and an account of progress on ML-driven macro placement are a 2021 release of a state-of-the-art commercial tool. More examples in Section 3.1.

The remaining standard-cell P&R flow is performed using a 2021 release of a state-of-the-art commercial tool. More examples and an account of progress on ML-driven macro placement are available from [47] and linked documents.

4 CLOUD-NATIVE ENABLEMENT OF LARGE-SCALE DESIGNS OF EXPERIMENTS

Machine learning requires data, hence ML EDA also requires data. However, in the EDA and IC design context, generating large amounts of data typically requires enormous compute resources and takes significant time. To enable efficient generation of the large volume of data needed to support ML EDA, our efforts focus on bringing attention to use of cloud computing resources which are now prevalent and easily accessible from commercial providers. Cloud-native enablement will help enable scalable CAD/EDA optimizations that leverage massive data and parallelism [12].

An important question is how to best utilize a public cloud to generate large-scale data for ML EDA. Merely obtaining bare cloud instances, installing libraries and compiling tools every time does not scale well and is not easily portable to alternate cloud/compute environments. This section describes a generic, cloud-native way to enable large-scale designs of experiments for ML-enabled EDA using public cloud services.

4.1 Container and Container Orchestration

A container is a software package which contains the application and all its dependencies to run the application. It isolates different environments, container orchestration is a standard and generic way to deploy, manage, and scale-out containers on a large-scale compute infrastructure. Thanks to this abstraction of the compute infrastructure, container orchestration is a standard and generic way of deploying containerized applications regardless of the underlying compute infrastructure. Kubernetes [49] is the most widely used container orchestration platform. Here are a few key terminologies.

- A node is a compute machine, such as a physical server, a virtual machine, or a cloud instance.
- A set of nodes that are grouped together is called a cluster.
- A pod is a construct to run containers in a Kubernetes cluster, providing an isolated environment for containers in a node.
- Storage resources are abstracted by persistent volume (PV). A PV is external storage made available in a cluster, such as a network file system or cloud file storage.
We have a single pod that runs the application container, which can be a local machine, such as a laptop or a server. Since (local machine, laptop, private/public cloud, etc.).

In this subsection, we discuss use of Ray clusters to deal with the underlying compute fabric to build distributed applications.

2. Create a Docker image that can run on the Kubernetes cluster as containers and communicate with Ray.

3. Use PVs in pods, we need persistent volume claims (PVCs). A PVC makes a PV mountable to pods as their storage.

4. Kubernetes Cluster for Large-Scale Design of Experiments

Figure 9 shows a generic Kubernetes cluster suitable for large-scale designs of experiments (DoEs). It includes a set of nodes, each of which can be a local machine such as a laptop or a server. Since we also need storage for our data (PDK, libraries, and designs), we have a PV and its PVC in the cluster, which is associated with a storage element; this can be local storage or cloud-hosted storage. We have a single pod that runs the application container, which is the application of interest, e.g., a logic synthesis or P&R tool container. Note that Figure 9 is a cloud-native Kubernetes cluster, meaning that we can deploy it to any sort of compute environment (local machine, laptop, private/public cloud, etc.).

To enable large-scale design experiments, we want larger Kubernetes clusters that have more resources. One way to achieve the larger cluster is to scale-out the number of containers within the cluster. A natural subsequent question would be how to efficiently distribute multiple design experiments across different pods. For example, it is possible to issue a Kubernetes command multiple times to execute each experiment, but this will not scale well.

Here, we introduce Ray [29] as shown in Figure 9. Ray is a widely-used distributed execution framework developed by UC Berkeley. It provides a simple Python API to build distributed applications. Ray works well with Kubernetes via Ray Kubernetes Operator [44], and also supports other cloud-based clusters. Ray and Kubernetes can be used to enable distributed design experiments: we describe experiments using the Ray Python API, and deploy experiments across the Kubernetes cluster using Ray.

4.3 Example: Cloud-Native Design of Experiments with Kubernetes and Ray

To deploy a given design of experiments into the Kubernetes cluster shown in Figure 9, we create a Docker image that can run on the Kubernetes cluster as containers and communicate with Ray.

In last year’s RDF update [4], we described the use of Ray for flow autotuning [50]. In this subsection, we discuss use of Ray clusters to deal with the underlying compute fabric to build distributed applications.

Figure 9: Kubernetes cluster example for large-scale DoE. PV/PVC stores PDK and cell libraries, serves as common data storage, and is mounted on pods. Ray is used to distribute an individual experiment to the pods.

$FIGURE 9: Kubernetes cluster example for large-scale DoE. PV/PVC stores PDK and cell libraries, serves as common data storage, and is mounted on pods. Ray is used to distribute an individual experiment to the pods.$

Figure 10: Dockerfile template for deploying into Kubernetes cluster with Ray.

```python
import ray
import subprocess

@ray.remote(num_cpus=2)
def sweep_utilization(util):
    # 1. Copy experiment template
    template = "workspace/experiment-template"
    workspace = "workspace/experiment-{}/".format(util)
    subprocess.call("cp -r {} "workspace, shell=True)

    # 2. Change the utilization.
    with open("{}".format(workspace), 'a') as f:
        f.write("export CORE_UTILIZATION = {}\n".format(util))

    # 3. Execute the flow
    subprocess.call("cd {} && make DESIGN_CONFIG=./config.mk".format(workspace), shell=True)

    ray.init(address)
    obj_refs = [sweep_utilization.remote(_) for _ in range(40,60)]
    while True:
        if len(remain) == 0:
            break
```

Figure 11: An example OpenROAD DoE using Ray.

Figure 10 shows an example of such a Docker image for OpenROAD. We first create a dedicated Docker image for the application (Line 1). Here, we assume that the image is named openroad:openroad; in this way, we can utilize the original application’s Docker image as-is, and install additional packages to make it cloud-native. We then install the necessary packages to make it work with Ray (Lines 3–10). The packages include kubectl as well as Python modules kopf, kubernetes, and ray, as shown in Figure 10.

A Ray Python code example is shown in Figure 11, taking a maximum floorplan utilization design goal as an example. We import necessary packages (Lines 1–2) and define a function named sweep_utilization (Lines 4–18). By adding the @ray.remote decorator (Line 4), we can make the function distributable by Ray. The function copies a reference flow template (Lines 6–10). It then modifies the template to the desired utilization (Lines 12–14), and finally runs the flow (Lines 16–18). With the function defined, we can call the function in parallel with different utilizations (Lines 20–25). The argument address of ray.init() (Line 20) specifies the Kubernetes cluster URL, which can be obtained by Kubernetes commands. Once the address is registered to Ray, the distribution of multiple experiments is handled by Ray under the hood. Several detailed examples are available in our GitHub repository [45], which
provides full Kubernetes-Ray examples for finding the maximum utilization and the maximum clock frequency of a design.

5 CONCLUSION

This paper has described several recent developments in RDF-2022. (1) Existing RDF elements have been improved, as follows. The RosettaStone effort to bridge past academic contests and codes to modern designs and enables has been enhanced for improved timing: sensibility in benchmark netlists. METRICS2.1 infrastructure in OpenROAD and OpenROAD-flow-scripts now uses native JSON metrics reporting. Calibrations data has been augmented; moreover, recent examples of incorrect or “uncalibrated” use of open enables motivate increased attention to the issue of calibration. (2) Multiple efforts have focused on macro placement. (i) New open-source benchmarks on open PDKs, with corresponding flows for fully reproducible results, give improved baselines for academic research. (ii) The macro placement step has been explicitly added into the RDF-2022 flow, along with the RTL-MP and Hier-RTLMP engines. (iii) We describe potential use of these additions to RDF in both ML-aided and ML-driven macro placement research. (3) We also present an approach to establishing a generic, cloud-native large-scale design of experiments for ML-enabled CAD.

One area of future work is the addition of more Designs of Experiments in the Metrics4ML repo [30], with open ML model-building and prediction challenges for the research community. An example would be the data-driven modeling and prediction of congestion and routed wirelength from structural netlists. Another important direction is enhancement of the MacroPlacement repo [31] – and RDF overall – to include additional open-source designs and enables (e.g., GF180MCU [46]), and to understand more clearly how ML can further improve the achievable PPA for relevant testcases. The MacroPlacement effort also highlights the importance of filling in gaps such as open-source memory generators or “fake-stack” alternate BEOL generators, which are typically not available with open PDKs and cell libraries but are crucial to address relevant research questions.

6 ACKNOWLEDGMENTS

This work is supported in part by the IEEE Council on Electronic Design Automation (CEDA), by DARPA HR0011-18-2-0032, and by NSF CCF-2126665.

REFERENCES


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


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


[27] SKY130-PDK. https://github.com/google/skywater-pdk


[34] NVidia open source hardware. https://github.com/nvidia/nv-tPHY


[38] M. Venn, “MPW1 silicon arrived! What went wrong?”. https://www.youtube.com/watch?v=LwVucqGjkdE


[41] Skywater open source PDK. https://github.com/google/skywater-pdk


[47] A. B. Kahng, "For the Record - An Update", August 2022. https://docs.google.com/document/d/1-cwavoDE9WCzyBzAdNqCq9rOAhKq1vFVI0Y_48FCY/edit


