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-sensible 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:
Jinwook Jung, Andrew B. Kahng, Ravi Varadarajan, and Zhiang Wang. 2022. IEEE CEDA DATC: Expanding Research Foundations for IC Physical Design and ML-Enabled EDA: Invited Paper. In *ICCAD'22: IEEE/ACM 2022 International Conference On Computer Aided Design, October 30–November 03, 2022, San Diego, CA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3508352.3561379

ICCAD'22, October 30–November 03, 2022, San Diego, CA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3561379

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-sensible 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

Table 1: RDF-2022 Components

Component	Tools
RTL generator	Chisel/FIRRTL
RTL obfuscation	ASSURE
Logic synthesis	Yosys, ABC
DFT insertion	Fault
Floorplanning	TritonFP
Macro Placement	TritonMP, **RTL-MP**, **Hier-RTLMP**
Global placement	RePlAce, FZUplace, NTUPlace3, ComPLx, Eh?Placer, FastPlace3-GP, mPL5/6, Capo
Detailed placement	OpenDP, MCHL, FastPlace3-DP
Flip-flop clustering	Mean-shift, FlopTray
Clock tree synthesis	TritonCTS
Global routing	FastRoute4-lefdef, NCTUgr, CUGR
Detailed routing	TritonRoute, NCTUdr, DrCU
Layout finishing	KLayout, Magic
Gate sizing	Resizer, TritonSizer
Parasitic extraction	OpenRCX
STA	OpenSTA, iTimerC
Database	OpenDB
Libraries/PDK	NanGate45, SKY130, ASAP7, **GF180MCU**, NCTUcell
Integrated app	OpenROAD
Benchmark conversion	**RosettaStone (timing-sensible)**

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-RTLMP* [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-sensibility 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-sensibility in *RosettaStone* (Section 2.1), native JSON-based deployment of METRICS2.1 (Section 2.2), and amplification of the *Calibrations* effort (Section 2.2).

2.1 Timing-Sensibility 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.

Table 2: Effect of logic cutting on timing sensibility of DAC2012 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.

Design	PreLC ECP (ns)	PreLC #Stages on WTP	PostLC ECP (ns)	PostLC #Stages on WTP
superblue2	8.55	327	2.47	11
superblue3	13.28	495	1.89	24
superblue5	7.26.	271	2.62	26
superblue6	7.18	267	2.67	26
superblue7	5.42	211	2.87	10
superblue9	8.14	308	2.05	82
superblue11	70.47	2497	2.37	27
superblue12	3.91	141	3.11	20
superblue14	16.76	450	2.78	26
superblue16	4.72	144	1.31	21
superblue19	7.25	279	1.79	18

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. All the benchmarks are implemented in the ASAP7 enablement [37]. Timing results are measured for the unplaced netlists using OpenSTA. We follow the convention seen in commercial tools, and ignore timing paths that pass through nets that have fanout larger than or equal to 100. Additional experimental results on other benchmarks are available in the repo [40], which also includes scripts that implement the path cutting flow.

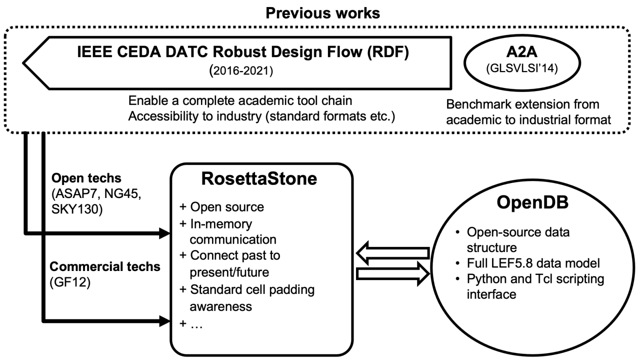


Figure 1: RosettaStone completes the vision of previous research enablements (RDF [4] and A2A [14]). Figure adapted from [13].

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"*


```
{
    "detailedroute__route__net": 78790,
    "detailedroute__route__net__special": 2,
    "detailedroute__route__drc_errors__iter:1": 54267,
    "detailedroute__route__wirelength__iter:1": 384221,
    "detailedroute__route__drc_errors__iter:2": 13959,
    "detailedroute__route__wirelength__iter:2": 378133,
    "detailedroute__route__drc_errors__iter:3": 13053,
    "detailedroute__route__wirelength__iter:3": 376415,
    "detailedroute__route__drc_errors__iter:4": 19,
    "detailedroute__route__wirelength__iter:4": 376298,
    "detailedroute__route__drc_errors__iter:5": 0,
    "detailedroute__route__wirelength__iter:5": 376297,
    "detailedroute__route__drc_errors": 0,
    "detailedroute__route__wirelength": 376297,
    "detailedroute__route__vias": 577573,
    "detailedroute__route__vias__singlecut": 577573,
    "detailedroute__route__vias__multicut": 0
}
```


Figure 2: OpenROAD metrics from detailed route.


```
{
    "cts__timing__setup__tns__pre_repair": -30.4700,
    "cts__timing__setup__ws__pre_repair": -0.2600,
    "cts__design__io__pre_repair": 388,
    "cts__design__instance__count__pre_repair": 15651,
    "cts__design__instance__area__pre_repair": 26229.7285,
    "cts__design__instance__utilization__pre_repair": 0.4969,
    "cts__timing__setup__tns__post_repair": -16.0600,
    "cts__timing__setup__ws__post_repair": -0.1200,
    "cts__design__io__post_repair": 388,
    "cts__design__instance__count__post_repair": 16431,
    "cts__design__instance__area__post_repair": 27251.1680,
    "cts__design__instance__utilization__post_repair": 0.5163,
}
```


Figure 3: Pre- and post-optimization metrics reporting obtained through use of metrics modifiers.

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 SKY130HD enablement. Here, the golden commercial enablement, used by SkyWater Technology with its commercial customers, is S8 [28]; the corresponding open-source SKY130 PDK [27] was derived from S8. The OpenROAD use of OpenRCX for SKY130 [26] 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

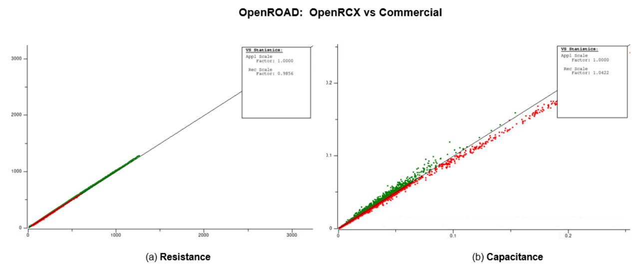


Figure 4: RC correlation between OpenROAD OpenRCX on SKY130HD open enablement [27], and commercial RCX tool on S8 commercial enablement [28].

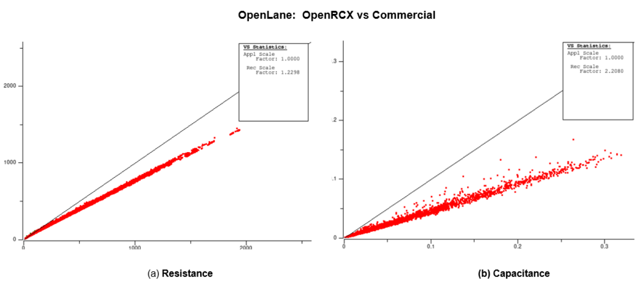


Figure 5: RC correlation between OpenLane OpenRCX on SKY130HD open enablement [27], and commercial RCX tool on S8 commercial enablement [28].

OpenRCX extractor, such that correlation has degraded substantially as seen in Figure 5. [38] describes another example involving incorrect use of static timing analysis. We believe this highlights a need for added attention to the DATC's Calibrations effort [51].

3 MACRO PLACEMENT

Macro placement is an important challenge for both IC physical design and ML-enabled EDA. In IC physical design, it is well-understood that the placement of macros in the floorplan will significantly impact final design QoR. Moreover, in today's physical synthesis flows (e.g., Synopsys Fusion Compiler or Cadence Genus iSpatial), a floorplan .def with macro and pin placements is typically needed as an input to the front-end physical synthesis. From the perspective of ML-enabled EDA, the scale of a macro placement problem instance is much smaller than that of standard cell P&R, which makes the related ML models much easier to train and understand. Thus, it is important to establish research foundations for macro placement. Toward this goal, progress includes the following. (i) We have established a modern set of open benchmarks for macro placement, and provided fully reproducible and reliable flows for these benchmarks. (ii) A new macro placer, *Hier-RTLMP*, can handle modern designs with complex logical hierarchies and hundreds to thousands of macros. This can serve as a baseline for other macro placers. (iii) We note implications for research directions in ML-enabled macro placement.

3.1 Modern Benchmarks for Macro Placement

Improvement of optimizers, including EDA tools, requires *relevant* and *probative* testcases or benchmarks. Often, "classical" benchmarks are outdated and/or incomplete – e.g., placement benchmarks in Bookshelf format do not have well-formed functional or timing information – and cannot be used for flow-scale evaluation. To address this, modern and fully-formed benchmarks are needed. The *MacroPlacement* repo [31] provides open-source designs in open enablements, along with multiple EDA tool flows.

Open-source designs include the following.

- *Ariane* [32] is a 64-bit RISC-V CPU design. *Ariane_136* contains 136 macros, and is generated by instantiating 16-bit memories. *Ariane_133* contains 133 macros, and is generated by updating the memory connections of the *Ariane_136* design.
- *MemPool* [33] is a many-core system targeting image processing applications. It implements 256 RISC-V cores that can access a large, shared L1 memory in at most five cycles.
- *NVDLA* (NVIDIA Deep Learning Accelerator) [34] is a free and open architecture that promotes a standard way to design deep learning inference accelerators.

Open-source enablements include the following.

- *NanGate45* Open Cell Library based on FreePDK45 [35], with the bsg_fakeram black-box SRAM generator [36] used to generate fake single-port SRAMs (i.e., LEF abstracts and Liberty models for P&R).
- *ASAP7* [37] 7nm predictive PDK, with FakeRAM2.0 [39] used to generate fake single-port SRAMs.
- *SKY130HD* [41], with the metal stack extended to nine layers since the original five-layer metal stack is insufficient to route macro-dominated testcases. [36] is used to generate fake single-port SRAMs.

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 "irreproducibility 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

[1]These 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.

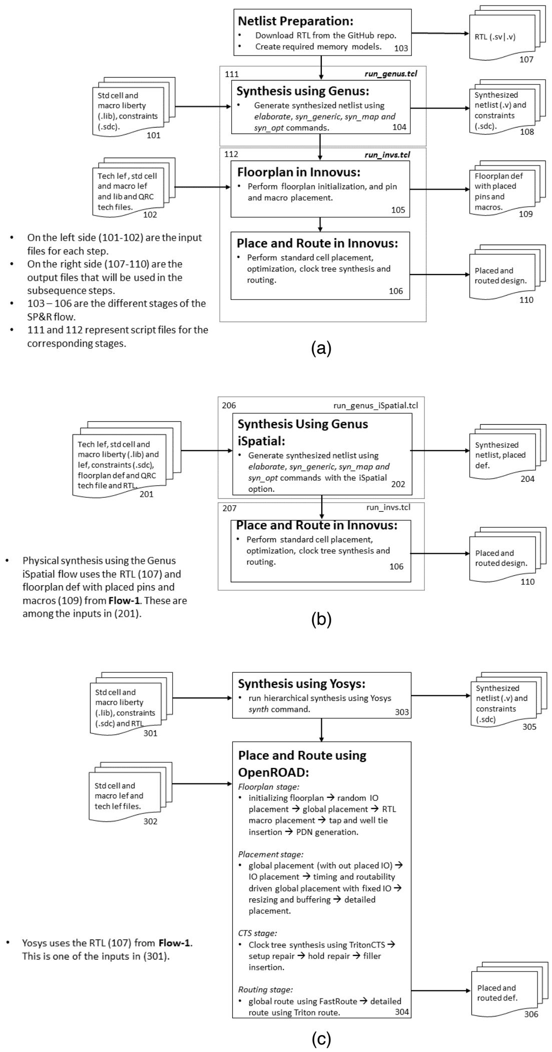


Figure 6: Multiple flows (a)–(c) provided by the *MacroPlacement repo* [31].

modified *ParquetFP* [42] to pack and snap macros into corners of the layout. *TritonMacroPlace* has been replaced by *RTL-MP* [17] which combines several key elements to produce human-quality macro placements: (i) exploitation of RTL logical hierarchy and autoclustering to transform the original netlist into a clustered netlist; (ii) sequential graph analysis and capture of dataflow using virtual connections between physical clusters of the clustered netlist; and (iii) comprehension of pin access, notch region avoidance and placement guidance for macros in its placement cost function.

However, *RTL-MP* cannot handle very large complex IP blocks, as its clustered netlist corresponds to a single-level physical hierarchy, and macros are constrained to be placed along the block periphery. A recently-developed *Hier-RTLMP* [16] escapes this limitation by adopting a multilevel methodology and new features that include the following. (i) *Hier-RTLMP* converts the logical hierarchy of the netlist into a multilevel physical hierarchy, which allows handling of large-scale complex designs and placement of macros in the core area. The dataflow of the design is captured through selectively merging and dissolving logical modules in the logical hierarchy. (ii) Initial shape functions for physical clusters are calculated bottom-up from leaf clusters, and then a given cluster's shape is refined during top-down hierarchical macro placement as the location and shape of its parent cluster are determined. This two-step shaping process leads to improved runtime and QoR outcomes. (iii) To avoid global congestion issues, a hierarchical bus planning engine performs route planning for the global nets at each level of the physical hierarchy, to determine the pin access regions for each of the physical clusters.

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 $tabla\_02$ with 760 macros, from the open-source project [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 unit clusters. 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.

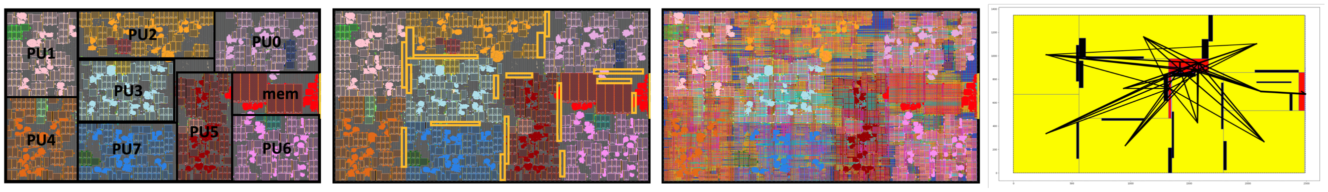
3.3 ML-Driven Macro Placement

Machine learning approaches have been applied to different stages of physical design, such as standard cell placement [1] and clock tree synthesis [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 el al. [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 [17]

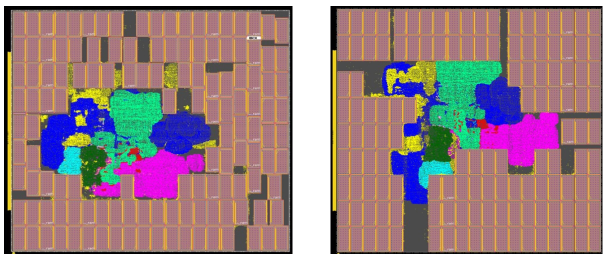
$$cost = \alpha \times Area + \beta \times WL + \gamma \times p_{outline} + \zeta \times p_{bias} + \eta \times p_{blockage} + \theta \times p_{guidance} + \lambda \times p_{notch} \quad (1)$$

where $Area$ is the area of the current floorplan, W is a wirelength estimate (HPWL), $p_{outline}$ is the penalty for violating the fixed



(a) (b) (c) (d)

Figure 7: *Hier-RTLMP* results for the `tabla_02` design [5]. (a) Post-placement layout. (b) Post-placement layout with highlighted pin access regions. (c) Post-routing layout. (d) Top-level bus planning result.



(a) (b)

Figure 8: Post-routing layouts of design *Ariane-133* [32] implemented in NanGate45 [35]. The macro placements in (a) and (b) are created respectively by the Circuit Training RL-based approach [19] [43] and by *Hier-RTLMP*.

outline constraint, p_{bias} is the penalty to promote macro peripheral bias, $p_{blockage}$ is the penalty for pin access and macro blockage, $p_{guidance}$ is the penalty for macro guidance, p_{notch} is the penalty for notch regions, and α, β, γ, ζ, η, θ, λ are the corresponding weights. Currently, the hyperparameter tuning tool *Tune* [29] is used for automatic design-specific weight tuning. On the other hand, the work of [1] suggests that RL-based parameter tuning might better optimize the weights within less runtime. Since all codes and algorithms of *Hier-RTLMP* are available, a trained RL model based on *Hier-RTLMP* should have better interpretability, compared to any that are based on commercial tools.

3.3.2 ML-driven Macro Placement. We may also train ML models to perform macro placement from scratch. In this vein, the Google Brain reinforcement learning-based approach [19] to macro placement has stimulated a great deal of interest across academia and industry. As part of a broad discussion of the method and its replication, we have released the *MacroPlacement repo* [31], which provides macro-heavy open-source benchmark designs in open enablements (see Section 3.1), along with implementations of missing or binarized code elements from [43]. This enables the RL-based approach of [19] [43] to be run on the macro placement benchmarks in Section 3.1.

Figure 8 shows example macro placement results of the *Ariane-133* design [32] implemented in NanGate45 [35]. Figure 8(a) shows the macro placement generated by the Google RL-based approach, while Figure 8(b) shows the macro placement generated by *Hier-RTLMP*. 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 applications from each other, while sharing the same OS kernel. A *containerized application* comes with all its dependencies but does not include the full OS, avoiding latency or performance overhead issues.

A *container orchestration service* abstracts the underlying compute infrastructure, which can be on-premises datacenter, local servers, public cloud services, or even a mixture of those. It is used to deploy, manage, and scale-out containers on a large-scale compute infrastructure. Thanks to this abstraction of the compute environment, 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.

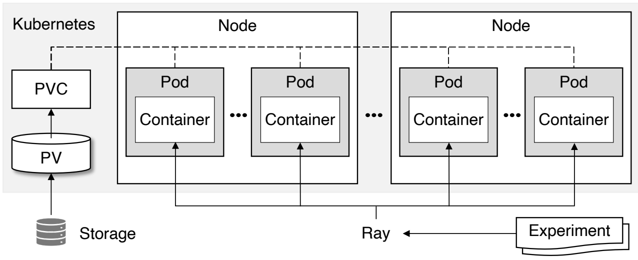


Figure 9: Kubernetes cluster example for large-scale DoE. PV/PVC stores PDK and cell libraries, serves as common data storage, and is mounted on pods. Ray is used to distribute an individual experiment to the pods.

- To use PVs in pods, we create *persistent volume claims* (PVCs). A PVC makes a PV mountable to pods as their storage.

4.2 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.[2] 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.[3]

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.

[2]It is possible to have multiple PVs and corresponding PVCs, each of which may contain PDK, libraries, and designs separately.

[3]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.

```
1  FROM openroad/openroad
2
3  # For kubectl
4  RUN yum install -y kubectl
5
6  # For Ray
7  RUN yum -y update \
8      && yum install -y python39 kubectl \
9      && alternatives --set python /usr/bin/python3 \
10     && python -m pip install kopf kubernetes ray
```

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

```
1  import ray
2  import subprocess
3
4  @ray.remote(num_cpus=2)
5  def sweep_utilization(util):
6    # 1. Copy experiment template
7    template = "/workspace/experiment-template"
8    workspace = "/workspace/experiment-{}".format(util)
9    subprocess.call("cp -r {} {}".format(template, workspace),
10                   shell=True)
11
12   # 2. Change the utilization.
13   with open("{}/config.mk".format(workspace), 'a') as f:
14     f.write("export CORE_UTILIZATION = {}\\n".format(util))
15
16   # 3. Execute the flow
17   subprocess.call("cd {} && make DESIGN_CONFIG=./config.mk"
18                   .format(workspace), shell=True)
19
20 ray.init(address)
21 obj_refs = [sweep_utilization.remote(_) for _ in range(40,60)]
22 while True:
23     _, remain = ray.wait(obj_refs)
24     if len(remain) == 0:
25         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 enablements 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 enablements 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 enablements (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-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. Chen, I. H.-R. Jiang, J. Jung, A. B. Kahng, V. N. Kravets, *et al.*, "DATC RDF-2019: towards a complete academic reference design flow", *Proc. ICCAD*, 2019, pp. 1–6.
[3] J. Chen, I. H.-R. Jiang, J. Jung, A. B. Kahng, V. N. Kravets, *et al.*, "DATC RDF-2020: strengthening the foundation for academic research in IC physical design", *Proc. ICCAD*, 2020, pp. 1–6.
[4] J. Chen, I. H.-R. Jiang, J. Jung, A. B. Kahng, S. Kim, *et al.*, "DATC RDF-2021: design flow and beyond", *Proc. ICCAD*, 2021, pp. 1–6.
[5] H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, A. B. Kahng, *et al.*, "VeriGOOD-ML: an open-source flow for automated ML hardware synthesis", *Proc. ICCAD*, 2021, pp. 1–7.
[6] 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.
[7] 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", *Proc. Workshop on Open-Source EDA Technology*, 2018, pp. 1–4.
[8] D. Junkin, "Supporting the scientific method for the next generation of innovators", Open-Source EDA and Benchmarking Summit *ACM/IEEE DAC birds-of-a-feather session*, July 2022. https://open-source-eda-birds-of-a-feather.github.io/doc/slides/BOAF-Junkin-DAC-Presentation.pdf
[9] J. Jung, A. B. Kahng, S. Kim, and R. Varadarajan, "METRICS2.1 and flow tuning in the IEEE CEDA robust design flow and OpenROAD", *Proc. ICCAD*, 2021, pp. 1–9.
[10] J. Jung, I. H.-R. Jiang, J. Chen, S.-T. Lin, Y.-L. Li, *et al.*, "DATC RDF: an academic flow from logic synthesis to detailed routing", *Proc. ICCAD*, 2018, pp. 1–4.
[11] J. Jung, I. H.-R. Jiang, J. Chen, S.-T. Lin, Y.-L. Li, *et al.*, "DATC RDF: an open design flow from logic synthesis to detailed routing", *Proc. Workshop on Open-Source EDA Technology*, 2018, pp. 1–4.
[12] A. B. Kahng, "Machine learning for CAD/EDA: the road ahead", *IEEE Design & Test* (2022).
[13] A. B. Kahng, M. Kim, S. Kim and M. Woo, "RosettaStone: Connecting the past, present and future of physical design research", *IEEE Design & Test* (2022).
[14] A. B. Kahng, H. Lee and J. Li, "Horizontal benchmark extension for improved assessment of physical CAD research", *Proc. GLSVLSI*, 2014, pp. 27–32.
[15] A. B. Kahng and S. Mantik, "A system for automatic recording and prediction of design quality metrics", *Proc. ISQED*, 2001, pp. 81–86.
[16] A. B. Kahng, R. Varadarajan and Z. Wang, "Hier-RTLMP: A hierarchical automatic macro placer for large-scale complex IP blocks", *draft in submission*, 2022.
[17] A. B. Kahng, R. Varadarajan and Z. Wang, "RTL-MP: toward practical, human-quality chip planning and macro placement", *Proc. ISPD*, 2022, pp. 3–11.
[18] Y.-C. Lu, J. Lee, A. Agnesina, K. Samadi and S. K. Lim, "GAN-CTS: a generative adversarial framework for clock tree prediction and optimization", *Proc. ICCAD*, 2019, pp. 1–8.
[19] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, *et al.*, "A graph placement methodology for fast chip design", *Nature* 594 (2021), pp. 207–212. https://github.com/google-research/circuit_training
[20] T. Spyrou, "OpenDB, OpenROAD's database", *Proc. Workshop on Open-Source EDA Technology*, 2019.
[21] IEEE CEDA design automation technical committee. https://ieee-ceda.org/node/2591.
[22] DATC robust design flow. https://github.com/ieee-ceda-datc/datc-rdf
[23] The OpenROAD project. https://github.com/The-OpenROAD-Project
[24] The OpenROAD flow. https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts
[25] OpenLane https://github.com/The-OpenROAD-Project/OpenLane
[26] OpenROAD SKY130HD RCX rules file. https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/blob/master/flow/platforms/sky130hs/rcx_patterns.rules
[27] SKY130-PDK. https://github.com/google/skywater-pdk
[28] S8 PDK V2.0.1. https://foss-eda-tools.googlesource.com/skywater-src-nda/+/refs/heads/master/s8/V2.0.1/
[29] Ray. https://www.ray.io
[30] Metrics4ML. https://github.com/ieee-ceda-datc/datc-rdf-Metrics4ML
[31] Macro placement. https://github.com/TILOS-AI-Institute/MacroPlacement
[32] Ariane RISC-V CPU. https://github.com/lowRISC/ariane
[33] MemPool. https://github.com/pulp-platform/mempool
[34] NVDLA open source hardware. https://github.com/nvdla/hw/tree/nv_small
[35] FreePDK45. https://eda.ncsu.edu/downloads/
[36] BSG black-box SRAM generator. https://github.com/jjcherry56/bsg_fakeram
[37] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, et al., "ASAP: a 7-nm FinFET predictive process design kit", *Microelectronics Journal* (53) (2016), pp. 105–115. https://github.com/The-OpenROAD-Project/asap7
[38] M. Venn, "MPW1 silicon arrived! What went wrong?", https://www.youtube.com/watch?v=lw9ucvgQJjk&t=413s
[39] FakeRAM2.0. https://github.com/ABKGroup/FakeRAM2.0
[40] RosettaStone. https://github.com/ABKGroup/RosettaStone
[41] SkyWater open source PDK. https://github.com/google/skywater-pdk
[42] "ParquetFP". http://vlsicad.eecs.umich.edu/BK/parquet
[43] Google research "circuit training". https://github.com/google-research/circuit_training
[44] Ray Kubernetes operator. https://ray-project.github.io/kuberay/components/operator/
[45] Kubernetes-based large-scale DoE. https://github.com/ieee-ceda-datc/datc-k8s-doe
[46] GlobalFoundries GF180MCU open source PDK. https://github.com/google/gf180mcu-pdk
[47] A. B. Kahng, "For the Record - An Update", August 2022. https://docs.google.com/document/d/1c-uweo3DHiCWZyBzAdNCqqcOrAbKq1sVIfY0_4bFCYE/edit
[48] OpenSTA : parallax static timing analyzer. https://github.com/The-OpenROAD-Project/OpenSTA/tree/937efddf5ecade9b927829074ab3bb274290a949
[49] Kubernetes. https://kubernetes.io/
[50] DATC RDF AutoTuner. https://github.com/ieee-ceda-datc/datc-rdf-flow-tuner
[51] DATC RDF calibrations. https://github.com/ieee-ceda-datc/datc-rdf-calibrations