An Updated Assessment of Reinforcement Learning for Macro Placement

Chung-Kuan Cheng, *Fellow, IEEE*, Andrew B. Kahng, *Fellow, IEEE*, Sayak Kundu, *Student Member, IEEE*, Yucheng Wang, *Student Member, IEEE*, and Zhiang Wang, *Student Member, IEEE*

***Abstract*—We provide an improved assessment of Google Brain's deep reinforcement learning approach to macro placement [29] and its updated Circuit Training (*CT*) implementation in GitHub [53]. A stronger simulated annealing (*SA*) baseline leverages the "go-with-the-winners" metaheuristic [3] and a multi-threading implementation. We develop and release new public benchmarks in sub-10nm technology: LEF/DEF for Google's 7nm TSMC Ariane protobuf and scaled variants, as well as testcases implemented in the open-source ASAP7 7nm research enablement. We evaluate from-scratch training and fine-tuning results for the latest "AlphaChip" release of Circuit Training, alongside multiple alternative macro placers. We also study the recently-published pre-training guidance in [53]. A commercial place-and-route tool is used to provide "true reward" post-route power, performance and area metrics. All data, evaluation flows and related scripts are publicly available in the *MacroPlacement* GitHub repository [63]. Our study affords insights into reproducibility and reporting in the research literature, and points out still-missing confirmations (e.g., of *CT*'s scalability and pre-training methodology) that remain open questions for the research community.**

I. Introduction

Macro placement is a fundamental problem in VLSI physical design that involves determining the positions of large circuit blocks (macros) on a chip layout canvas. These macros typically include memory arrays, processor cores, analog blocks, and other pre-designed components that are significantly larger than standard cells. The macro placement problem is NP-hard and involves complex trade-offs between multiple objectives including wirelength, area utilization, timing closure, power consumption, and routing congestion. The quality of macro placement directly impacts the final chip performance, manufacturability, and cost, making it one of the most critical steps in modern chip design flows.

In June 2021, authors from the Google Brain and Chip Implementation and Infrastructure (CI2) teams reported a novel reinforcement learning (RL) approach for macro placement [29] (*Nature*). The authors stated, "In under six hours, our method automatically generates chip floorplans that are superior or comparable to those produced by humans in all key metrics, including power consumption, performance and chip area." Results were reported to be superior to those of the RePlAce academic placer [5] and the simulated annealing (*SA*) metaheuristic.

Nature authors promised data and code availability, e.g., "The code used to generate these data is available from the corresponding authors upon reasonable request". The Circuit Training (*CT*) repository [53], which "reproduces the methodology published in the Nature 2021 paper", was made public in January 2022. However, reproduction and evaluation of *Nature* and *CT* has been hampered because neither the data nor the code used in these works is, as of November 2025, *fully* available. The irreproducibility of *Nature* claims has led to controversy and slowed progress in the field.

This situation reflects a broader scientific concern with reproducibility and integrity. When high-profile claims shape research agendas and public perception, transparent and replicable evaluation becomes essential. Recent discussions in *Nature* (Jan. 2025) [33] and work in *PNAS* (Aug. 2025) [30] emphasize this point. For macro placement specifically, broad coverage in mainstream and trade press of RL-based approaches after *Nature* underscores the EDA community's responsibility to deliver clear, technically rigorous, and reproducible assessments of empirical merit and practical impact.

A 2022 "Stronger Baselines" (*SB*) study, performed at Google [4], claimed that a properly-implemented simulated annealing outperforms *Nature*. [4] used a Google-internal version of *CT* with different benchmarks and evaluation metrics. Our conference paper [6] presented efforts toward an open-source implementation and assessment of *Nature* and *CT*. We made the data, scripts and results public in the *MacroPlacement* GitHub repository [63], with additional background, details, FAQs, etc. given in updates [68], an extended arXiv version [7], and "Our Progress" [64] and other documentation in [63].

In 2024, *Nature* authors published an addendum [14] to clarify methods and results of [29]. Importantly, updates to *CT*, dubbed "AlphaChip", were publicly released with pre-trained model weights (CT-AC) [53].[1] Markov [27] published a peer-reviewed meta-analysis of the *Nature* results, cross-checking [4] and [6]. Goldie et al., in the arXiv post [13], criticized [6], raising issues such as: (i) absence of peer-review; (ii) lack of RL pre-training; (iii) potential non-convergence of training for some testcases; (iv) use of 45nm and 12nm technologies versus sub-10nm (TSMC 7nm, corresponding to TPU v4) in *CT*; and (v) insufficiency of compute resources used.

Against this backdrop, rather than proposing a new algorithm, we carefully evaluate the extent to which the *Nature* method improves upon *prior approaches*, using extensive experimentation and a transparent methodology. In this work, we re-execute and augment the studies reported in [6] to thoroughly and conclusively address the criticism in [13]. Through this effort, we obtain a more rigorous assessment of the *CT* approach, along with further insights into report-

[1] Below, we refer to training Circuit Training (commit hash: 4c6fd98) from scratch as *CT-Scratch*. We denote fine-tuning of "AlphaChip" using the checkpoint released in August 2024 (commit hash: 4c6fd98) as *CT-AC*.

ing and reproducibility in the research literature. Our main contributions beyond [6] are as follows.

- **Evaluation of updated CT.** For all testcases, we train the updated Circuit Training from scratch, and also fine-tune AlphaChip from Google's August 2024 pre-trained checkpoint. Experimental observations, including resource requirements, model quality, and convergence and variability behavior, are reported in Sections VI and VII below.
- **Improved SA.** We strengthen the simulated annealing baseline by incorporating multi-threading and a 1994 "go-with-the-winners" [3] metaheuristic, while also ensuring reproducibility of executions (see Section IV). The improved *SA* achieves up to 26% better proxy cost within the same runtime while using only a quarter of the resources, compared to the *SA* implementation reported in [6]. The improved SA maintains superior performance over recent *CT-AC* methods, reaffirming the effectiveness of carefully optimized classical heuristics for combinatorial optimizations in the context of macro placement.
- **Sub-10nm testcases.** We strengthen sub-10nm experimental enablement in two ways. (i) We convert Google's public TSMC 7nm Ariane testcase (*CT-Ariane*) from protobuf [53] to LEF/DEF; we publish this and additional scaling studies of macro placement optimizers, following the "quantified scaling suboptimality" methodology of [17] and evaluating AlphaChip's performance on "blocks with over 500 macros" [14] in the *MacroPlacement* repository [63]. (ii) We port our testcases to a second 7nm enablement, the open-source academic ASAP7 PDK from ASU/Arm [41]. Studies with these new sub-10nm enablements reaffirm findings of [6].
- **Pre-training studies.** We perform pre-training of *CT* following the instructions in the Circuit Training repository [53]. Our studies (see Section VII) highlight the need for further confirmations of scalability, resource efficiency and other claims in [29].
- **Addressing resources and convergence.** Our revised experimental protocol provides compute resources that are sufficient for *CT* per [29] [53]. Furthermore, in both training from scratch (*CT-Scratch*) and fine-tuning (*CT-AC*) – for all our testcases – we double iterations from 200 to 400 to provide sufficient opportunity for *CT* to converge, and conduct multiple trials before declaring non-convergence.

Our updated and strengthened evaluation reconfirms conclusions of [6]. Our contribution is evidence-based evaluation – rigorous experiments, strengthened baselines, and reproducible sub-10nm enablements – rather than a new placement algorithm; this prioritizes clarity on empirical merit and practical impact. The simulated annealing and human baselines continue to show superiority to the latest AlphaChip while using substantially fewer resources. Further, *scaled* sub-10nm Ariane variants expose additional weaknesses of *Nature*, e.g., regarding stability, scalability, and resource demands. The *MacroPlacement* effort highlights the importance of "frictionless reproducibility" [11], along with open source code and data releases "upon which others then build" [31], in the academic EDA field and its nexus with AI/ML.

In the following, Section II lists the macro placement methods studied, Section III describes efforts toward open-source replication of *CT* and Section IV details our simulated annealing approach. Section V presents our experimental setup, and Sections VI and VII present results. Section VIII provides conclusions and directions for future research.

II. Macro Placement Methods

VLSI physical design researchers and practitioners have studied macro placement for well over half a century, as reviewed in [28] [36]. In this work, we study the following macro placement methods.

- **Circuit Training** [53] uses the RL approach to sequentially place macros. *CT* first divides the layout canvas into small grid cells, then uses placement locations along with hypergraph partitioning to group standard cells into standard-cell clusters (soft macros), to set up the environment. The RL agent then places macros one by one onto the centers of grid cells; after all macros are placed, force-directed placement is used to determine the locations of standard-cell clusters and calculate the *proxy cost*. Proxy cost is comprised of three metrics – wirelength, density, and congestion – which are proxies for routed wirelength, design density, and routing congestion. Lower values of these metrics imply better design quality.[2] Finally, the negative of the *proxy cost* is provided as the reward feedback to the RL agent.
  In this work, we use three variants of Google's RL approach: (i) training AlphaChip (i.e., the latest version of the renamed framework in the Circuit Training repository [53]) from scratch (denoted as *CT-Scratch*); (ii) fine-tuning AlphaChip using the pre-trained checkpoint released in August 2024 (denoted as *CT-AC*); and (iii) fine-tuning AlphaChip using our own checkpoint pre-trained with specific testcase variants (denoted as *CT-Ours*). Note that our present work uses Circuit Training [53] commit hash 4c6fd98 from February 2025, while our previous work [6] used commit hash 91e14fd from August 2022.
- **RePlAce** [5] [49] models the layout and netlist as an electrostatic system. Instances are modeled as electric charges, and the density penalty as potential energy. Instances are spread apart according to the gradient with respect to the density penalty. Note that our present work uses RePlAce from OpenROAD [2] [49], commit hash f02a3d4 from August 2024, which is the appropriate comparison; our previous work [6] used a specific standalone RePlAce chosen to match the "Stronger Baselines" study [50], and *Nature* used a standalone RePlAce from the OpenROAD project repository [47] [62], which was deprecated in January 2021.
- **CMP** is a state-of-the-art commercial macro placer from Cadence Design Systems, which performs concurrent macro and standard-cell placement. CMP has been available in Innovus place-and-route tool from 2019 versions onward. CMP results also serve as input to the Cadence Genus iSpatial physical synthesis tool. We include results of CMP in our experimental study (see Section VI).
- **Human-Expert** macro placements are contributed by individuals at IBM Research [57], ETH Zurich and UCSD [59];

[2] [6] provides a detailed description of these proxy cost components. An open-sourced implementation [65] reproduces the black-box implementation of the proxy cost in *CT*.

human-expert placements are one of the two baselines used by *Nature* authors [29].

- **Simulated Annealing (*SA*)** is the second baseline used by *Nature* authors, and is studied by both *Nature* and *SB*. Annealing is applied to place macros in the same grid cells as *CT* (see Section IV).

All testcases and results from the above methods, along with all scripts and codes where applicable, are publicly available in the *MacroPlacement* GitHub repository [63].[3] Based on permission from Cadence Design Systems, we are able to make public the Cadence runscripts used to obtain post-route power, performance and area (PPA) metrics – i.e., final chip metrics – from macro placement solutions. Licensed users of Cadence Genus 21.1 and Cadence Innovus 21.1 are able to fully replicate our results, through post-route PPA metrics.[4]

Last, we note that since 2021, many researchers from the machine learning and EDA communities have proposed various RL-based macro placement methods [9], [8], [15], [12], [16], [24], [23], [32], [37], [35], [39]. However, these works only show results on (non-real, old-node) physical design contest testcases (a practice that has drawn criticism from Google authors [13]), and do not report post-route PPA metrics. In [25], the authors combine Circuit Training with simulated annealing to handle rectilinear layouts, and further show the results on proprietary in-house testcases. Our ongoing outreach to the authors in this recent literature aims to draw more attention to *MacroPlacement* testcases, runscripts and evaluation methods, to spur further assessment and understanding of the RL approach.

III. Replication of Circuit Training

In this section, we discuss clarifications and reproduction in open source of *CT*. We first summarize key mismatches between *CT* and *Nature*. We then discuss the computing resources needed for studies of Circuit Training. As in [6], we are thankful to Google engineers for answering questions and for many discussions that helped our understanding of *CT* starting in April 2022.

A. Discrepancies between CT and Nature

From its outset, the *CT* GitHub repo has been stated to reproduce the methodology published in *Nature* [76]. Yet, several discrepancies between *CT* and the claims of *Nature* authors should be noted in the present context [6] [7].

- *Availability of code.* Two key "blackbox" elements, i.e., force-directed placement and proxy cost calculation, are neither clearly documented in *Nature* nor visible in *CT*. Reverse-engineering and replication in open-sourced C++ are discussed in [6] and Section IV below.
- *Need for pre-training. Nature* [29] does not show benefits from pre-training in its "Table 1" metrics. Rather, [29] only shows benefits (from the pre-trained model) in terms of runtime and final proxy cost. Additionally, the January 2022 ARIANE.md in Circuit Training [54] states that "Our results training from scratch are comparable or better than the reported results in the paper (on page 22) which used fine-tuning from a pre-trained model".
- *Gridding of macro placement locations.* The method described in *Nature* "place[s] the centre of macros and standard cell clusters onto the centre of the grid cells". However, *CT* does not require standard-cell clusters to be placed onto centers of grid cells.
- *Adjacency matrix construction.* The *Nature* paper describes generation of the adjacency matrix based on the *register distance* between pairs of nodes. This is a well-known technique (e.g., [34]) that is consistent with timing being a key metric for placement quality. However, *CT* builds its adjacency matrix based only on direct connections between nodes (i.e., macros, IO ports and standard-cell clusters).

For completeness, we recognize that approximately 2.5 years of effort and updates are embodied in the delta between *CT*'s commit hash 91e14fd studied in [6] and commit hash 4c6fd98 that we study here [53]. This delta has brought numerous changes to *CT*, spanning functionalities, default settings, and library dependencies. For example, (i) *CT* has now open-sourced pre-training and fine-tuning scripts, which we use in our present study. (ii) *CT* has also enabled use of DREAMPlace [26] to finalize soft macro placement. However, since our goal is to evaluate the claims in the original *Nature* paper – not newer variations that mix RL with other techniques already known to work well for macro placement – we do not use the DREAMPlace-enabled *CT* in our experiments. (iii) Parameter changes between the two commit hashes include reduction of gradient clipping from 1.0 to 0.1; reduction of the penalty for infeasible placement from -1 to -4; and reduction of #episodes per iteration from 1024 to 256 (which is stated to help with convergence and the final return). (iv) The hidden binary for *plc_client* has been updated from version 0.0.3 to 0.0.4, and our environment hence updates several library dependencies: TF-Agent 0.14.0 to 0.19.0, TensorFlow 2.10.0 to 2.15.0, dm-reverb 0.9.0 to 0.14.0, and CUDA 11.8 to 12.2.

B. Computing resources for Circuit Training

A critical concern [13] is whether compute resources are sufficient to enable assessment of Circuit Training and reproduction of the *Nature* results. Such resources have three main dimensions: training server; collect servers; and training iterations (equivalently, steps or walltime). To assess adequacy of resources, we rely on documentation from Google authors in, e.g., [29] [38] [53].

Training server. We use eight NVIDIA-V100 GPUs to train the model for global batch size = 1024. We believe this is adequate, based on [38] where the authors state, "We think the 8-GPU setup is able to produce better results primarily because it uses a global batch size of 1024, which makes learning more stable and reduces the noise of the policy gradient estimator. Therefore, we recommend using the full batch size suggested in our open-source framework in order to achieve optimal results." Circuit Training [53] itself shows the use of an 8-

[3] *MacroPlacement* [64] also includes macro placement solutions from AutoDMP [1] and Hier-RTLMP [21]. In this work, we do not compare with AutoDMP and Hier-RTLMP, as these were released after the *Nature* work.

[4] Note that the CT "grouping" step is implemented using the hMETIS binary, which is nondeterministic. Therefore, clustered netlists used in [6] are not reproducible, nor are studies of CT/AlphaChip that run the clustering step on a gate-level netlist. We describe below how use of the hMETIS shared library can avoid this nondeterminism (see Subsection VI-C).

GPU setup to reproduce their published Ariane results [54]. The global batch size = 1024 used in our runs is the same global batch size that is used in the Nature paper [29].[5]

Collect servers. In [38], Google authors write that "with distributed collection, the user can run many (10s-1000s) Actor workers with each collecting experience for a given policy, speeding up the data collection process." They further explain that "as mentioned in Section 2.2, data collection and multi-GPU training in our framework are independent processes which can be optimized separately."

Our previous work [6] used two collect servers each running 13 collect jobs, i.e., a total of 26 collect jobs were used for data collection. In our present work, we increase this to five collect servers each running at least 51 collect jobs, i.e., a total of 256 collect jobs are used for data collection. We believe this is adequate, again based on [38], which suggests that increasing collect jobs beyond this point has diminishing returns. (The *Nature* authors run 512 collect jobs for data collection, with the number of collect servers used to run these 512 collect jobs being unclear from the description provided. At the same time, [38] indicates that a larger number of collect jobs only speeds up training without affecting the outcome quality.) Since we use fewer collect jobs, our runs are slower, but quality is not compromised. We expect our runtimes to be higher than what Nature reports, and we define experimental protocols accordingly, as described next.

Training iterations. Published Google materials indicate a sufficient number of training iterations to use for *CT* in our experiments.

- *Train steps per second* is the indicator of the *CT* training speed. Figure 1's left plot shows the *CT* training speed of ∼0.9 steps/sec for Ariane [54] in our environment. The right plot shows the *CT* training speed (∼2.3 steps/sec) for *CT-Ariane* from a TensorBoard (no longer available) in the *CT* [54] repo. From this, we infer that our runtime is expected to be approximately 2.6× longer in our environment, compared to when the resources suggested in the *CT* repo are used.
- We observe that [54] gives 200 as a suggested number of iterations. To ensure adequacy of iterations afforded to *CT* in our experiments, we provide *another* 200 iterations, for a total of 400. Moreover, for larger testcases (BlackParrot Quad-Core and MemPoolGroup; see Table II), we extend further: in light of non-determinism in *CT* behavior, if training fails to converge after 400 iterations, we execute up to two additional 400-iteration runs before declaring non-convergence (i.e., "Divergence" in Tables III and IV).

IV. A Stronger Annealing Baseline

Both *Nature* and *SB* use simulated annealing (*SA*) [22] as a baseline for comparison. Table 2 of [4] gives a concise comparison of hyperparameters used by the two works. As in [6], we implement and run *SA* based on the description given in the *SB* manuscript. Our implementation differs from that

[5][29] refers to the use of 16 GPUs. However, based on the statements in [38] and what Circuit Training describes for "reproduce results", the final proxy cost achieved by our environment should not differ materially from the environment with 16 GPUs described in [29]. Indeed, using our setup we achieve similar proxy cost for Google's Ariane testcase (CT-Ariane) as reported in *CT* (see Subsection VI-A).

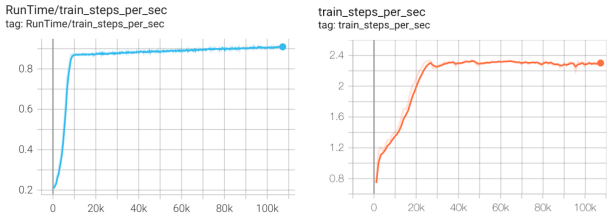


Fig. 1. Train steps per second plot for our CT run (left) and the CT run available (right) in the CT repository for Ariane.

described in *Nature* in its use of *move* and *shuffle* in addition to *swap*, *shift* and *mirror* actions. We also use two initial macro placement schemes, i.e., "spiral macro placement" whereby macros are sequentially placed around the boundary of the chip canvas in a counterclockwise spiral manner, and "greedy packer" whereby macros are packed in sequence from the lower-left corner to the top-right corner of the chip canvas [4]. Force-directed (FD) placement is used to update the locations of standard-cell clusters every $\{2n, 3n, 4n, 5n\}$ macro actions, where n is the number of hard macros; FD is not itself an action. The *SA* cost function is the proxy cost, which consists of wirelength, density and congestion.

As described in [6], Google's implementations of FD and proxy cost calculation are not open-sourced, but are only available via the *plc_client* in [53]. For speed and transparency, our *SA* experiments use our own C++ reimplementations of FD and proxy cost calculation; however, the Google *plc_client* is used for final FD soft macro placement and proxy cost evaluation at the end of the *SA* run. (Section 3.2 in [6] provides details of force-directed placement and proxy cost congestion.) Our *SA* codes are open-sourced in *MacroPlacement* [66].

A Go-With-the-Winners Enhancement. Similar to *Nature* and *SB*, our previous work in [6] runs 320 *SA* workers in parallel for 12.5 hours. Each worker, with its own hyperparameter setting, operates independently and does not communicate with other workers. Then, the macro placement solution with minimum proxy cost (as calculated by our C++ code) is used as the final *SA* solution. However, running 320 *SA* workers in parallel requires multiple servers, which may be impractical for users with limited computing resources.

To obtain a stronger *SA* baseline, we adopt the "go-with-the-winners" (GWTW) scheme [3] in a multi-threading implementation. In essence, GWTW allows a set of solution threads to proceed independently, but periodically executes a 'sync-up' whereby (i) the best threads are identified, (ii) their solutions are cloned to fill up the entire set of threads, and (iii) the threads then independently continue the solution process until the next 'sync-up'. This approach has seen previous adoption in physical design, e.g., for gate sizing [18].

The detailed algorithm is shown in Algorithm 1. The algorithm can be divided into following steps:

- **Lines 1-10:** We initialize *SA* workers in parallel, with each using a unique random seed to shuffle same-size macros. Placement is initialized via spiral initialization (resp. greedy packing) for workers with odd (resp. even) IDs.
- **Lines 14-17:** Each *SA* worker is run for $sync\_iter$ iterations in parallel. $sync\_iter$ is set based on $sync\_freq$. We use

$sync\_freq = 0.1$, meaning there are 9 synchronizations among the workers.

- **Lines 18-21:** The algorithm stops when $Iter$ iterations are performed; otherwise, $syncWorkers$ selects the top k workers based on proxy cost and replicates their macro locations and orientations to the remaining workers evenly.
- **Line 22:** writes out the best macro placement solution in terms of proxy cost for each worker.

Algorithm 1: Simulated Annealing

Input: Random seeds: $seed = 1$,
Number of iterations: $Iters$,
$N \times \#macro$ moves per iteration ($N = 20$),
Initial temperature: $T_0 = 0.005$,
Minimum temperature: $T_{min} = 1 \times 10^{-8}$,
Cooling rate: $\alpha = \exp\left(\frac{\ln(T_{min}/T_0)}{Iters}\right)$,
Number of workers: $W = 80$,
Replicated top $k = 8$ workers,
Synchronization frequency: $sync\_freq = 0.1$
Output: Macro placement solutions.

```
1  workers ← create W workers;
2  for i ← 0 to W − 1 in parallel do
3      workers[i].seed ← seed + i;
4      workers[i].N ← N;
5      workers[i].T ← T_0;
6      workers[i].α ← α;
7      if (i mod 2) = 0 then
8          workers[i].macro_placement ←
             "spiral macro placement";
9      else
10         workers[i].macro_placement ← "greedy packer";
11 iter_count ← 0;
12 sync_iter ← Iters × sync_freq;
13 while true do
14     end_iter ← min(Iters, iter_count + sync_iter);
15     for i ← 0 to W − 1 in parallel do
16         Each worker performs (end_iter − iter_count) SA
             iterations; applying N × #macro moves per
             iteration and updating temperature;
17     iter_count ← end_iter;
18     if iter_count = Iters then
19         break;
20     candidate_solutions ← extractTopK(workers, k);
21     Evenly distribute these top-k solutions across all the
         workers;
22 Write out the best solution of each worker.
```

As noted above, after placing soft macros (standard-cell clusters) with GWTW *SA*, we use *CT*'s *plc_client* to evaluate the proxy cost of the best macro placement solutions for each worker, and then return the best solution in terms of proxy cost for P&R evaluation. In our runs, the probabilities for five solution move operators (swap, shift, move, shuffle and flip) are respectively set to 0.24, 0.24, 0.24, 0.24 and 0.04. The number of iterations is set to ensure that overall runtime for each testcase is less than 12 hours on our slowest CPU server.[6]

Relative to the *SA* implementation in [6], our present *SA* implementation achieves similar or better results while using only one-fourth of the CPU resources: 80 threads instead of 320 threads, enabling execution on a single CPU server. Further, to ensure exact reproducibility across different platforms, (i) we use lookup tables for exponent computation and provide a binarized version of the lookup table in our repository; and (ii) we provide scripts to generate Docker and Singularity images that reproduce the same environment. Our testing across a range of Intel Xeon Gold and AMD EPYC CPUs confirms exact matching of *SA* solutions obtained by all 80 workers using the same Docker or Singularity image.

[6]For Ariane, BlackParrot, MemPoolGroup, Ariane-X2 and Ariane-X4, we set the number of iterations to 18K, 9K, 4.5K, 9K and 4K, respectively. This corresponds, e.g., to ∼11 hours on an Intel Xeon Gold 6148 CPU, or ∼3 hours on an AMD EPYC 9684X CPU.

TABLE I
GOOGLE'S TSMC 7NM ARIANE TESTCASE (*CT-ARIANE*) AND ITS SCALED VERSIONS. #GRPS INDICATES THE NUMBER OF STANDARD-CELL CLUSTERS.

Design	#StdCells (K)	#Grps	#Macros	#MacroType
CT-Ariane	83	799	133	1
CT-Ariane-X2	166	982	266	1
CT-Ariane-X4	332	1519	532	1

TABLE II
TESTCASES FOR EVALUATION. #FFS AND #MACROS RESPECTIVELY REPRESENT THE NUMBER OF FLIP-FLOPS AND MACROS IN BOTH NANGATE45 AND ASAP7.

Design	#StdCells (K)	#FFs (K)	#Macros	#MacroType
Ariane	99 - 117	20	133	1
BlackParrot	686 - 835	214	220	6
MemPoolGroup	2529 - 2729	361	324	4

V. EXPERIMENTAL SETUP

We now describe our experimental setup. We first describe testcases and design enablements. We then present the commercial evaluation flow used to evaluate macro placement solutions. Last, we present our settings for *CT*.

A. Testcases and enablements

To enable studies that are relevant to the sub-10nm regime [14], we develop scaled versions of Google's TSMC 7nm Ariane testcase (*CT-Ariane*), and port other macro-heavy testcases to the open 7nm enablement ASAP7 [41].[7] Details of testcases and design enablements used in our studies are as follows.

Testcases. We convert the only publicly available Google testcase, Ariane in TSMC 7nm (*CT-Ariane*), from protobuf format (available in the *CT* repo [53]) to LEF/DEF format [10]. Scaled (x2 and x4) versions of *CT-Ariane* serve as additional testcases, and are also used in pre-training and "quantified suboptimality" analyses [17]. Table I provides details of scaled versions of *CT-Ariane*.[8]

We also study three open-source testcases which are publicly available in [70]: Ariane [40], BlackParrot (Quad-Core) [42] and MemPoolGroup [46]. Table II provides testcase parameters. #MacroType gives the number of distinct macro sizes: Ariane has all same-sized macros, while BlackParrot and MemPoolGroup each contain macros of varying sizes. We use the 133-macro Ariane variant to match the Ariane in *Nature* and *CT*. Table II also gives ranges for standard cell counts, since #StdCells differs between NanGate45 and ASAP7.

[7]We drop the ICCAD04 [51] testcases, which correspond to much older technologies. Experimental results for these testcases remain available in [63].

[8]Note that the CT-Grouping flow uses hMETIS to generate standard-cell clusters with a parameter *npart*, which is set to 500 plus the number of predefined groups (macros and IO ports). Therefore, the number of standard-cell clusters (#Grps) scales sublinearly in Table I.

Enablements. Our studies use two open-source enablements that are public in *MacroPlacement*: NanGate45 [52] and ASAP7 [41]. We use the bsg_fakeram [43] generator to generate SRAMs for NanGate45. We also use FakeRAM2.0 [44] to generate SRAM abstracts for ASAP7-based testcases. We also use one closed-source enablement: GlobalFoundries 12LP with SRAMs from a third-party IP provider.

B. Commercial evaluation flow

Figure 2 presents the commercial tool-based flow that we use to create macro placement instances and evaluate macro placement solutions.[9] The flow has the following steps.

Step 1: We run logic synthesis using Cadence Genus 21.1 to synthesize a gate-level netlist for a given testcase.

Step 2: We input the synthesized netlist to Cadence Innovus 21.1 and use CMP (Concurrent Macro Placer) to place macros.

Step 3: We input the floorplan .def with placed macros to the Cadence Genus iSpatial flow and run physical-aware synthesis, which also generates initial placement locations (i.e., (x,y) coordinates) for all standard cells.

Step 4: We obtain macro placement solutions from seven methods: *CT-Scratch*, *CT-AC*, *CT-Ours*, *SA*, RePlAce, CMP and human-expert. The CMP macro placement is produced in Step 2. Before running *CT* or *SA* macro placement, we convert the Verilog netlist to protocol buffer (protobuf) format using code available in [63], and use CT-Grouping to generate standard-cell clusters. The initial placement of standard cells obtained in Step 3 is used to guide the CT-Grouping process [14]. For the *CT* and *SA* runs reported below, we run the grouping flow provided in Google's *CT* after generating the protobuf netlist. RePlAce and human experts are not given any initial placement information for standard cells or macros.

Step 5: For each macro placement solution, we input the floorplan .def with macro placement locations to Innovus for place and route (P&R). After reading the .def file into Innovus, we set all standard cells to unplaced, and legalize macro locations using the *refine_macro_placement* command.[10] We then perform power delivery network (PDN) generation.[11] After PDN generation, we run placement, clock tree synthesis, routing and post-route optimization (postRouteOpt).

Step 6: We extract the total routed wirelength (rWL), standard cell area, total power, worst negative slack (WNS), total negative slack (TNS) and DRC count from the post-routed design. Table 1 of the *Nature* paper [29] presents similar metrics to compare different macro placement solutions.[12] Below, we refer to these as the (*Nature*) "*Table 1* metrics".

[9]We do not perform any benchmarking of the EDA tools used in this study.

[10]Macro placements produced by RePlAce, *CT-Scratch*, *CT-AC*, *CT-Ours* and *SA* can have macros that are not placed on grids (cf. Subsection III-A).

[11]*CT* assumes that 18% of routing tracks are used by PDN [73]. We implement our PDN scripts following the "18% rule" for all the enablements. All of our PDN scripts are available at [74] in *MacroPlacement*.

[12]According to *Nature* authors, "The final metrics in Table 1 are reported after PlaceOpt, meaning that global routing has been performed by the EDA tool". In this paper, we report metrics after postRouteOpt, meaning that the entire P&R flow has been performed. A number of results reported in [64] include metrics after both PlaceOpt and postRouteOpt.

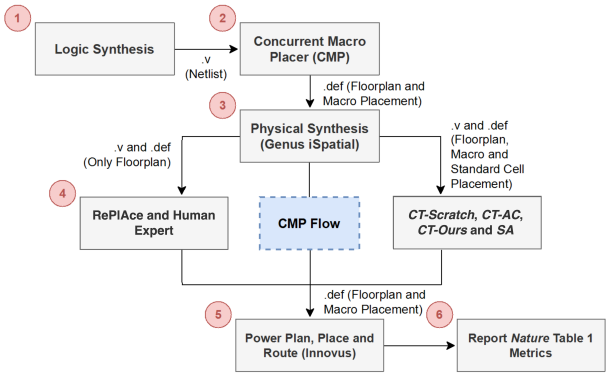


Fig. 2. Evaluation flow for placements produced by different macro placers.

C. Settings for Circuit Training

When running Circuit Training, we follow the default setting given in [53], except that we use a density weight of 0.5 instead of 1.0, based on guidance from Google engineers [56]. Training is conducted on a single server equipped with eight NVIDIA V100 GPUs, paired with five collect servers, each featuring a 96-thread CPU. Each training job deploys 256 collect jobs, as increasing beyond this threshold yields diminishing speedup benefits [29]. Unless otherwise mentioned, we use 333 as the global seed. As detailed in Subsection III-B, we use 200 iterations [77] for training and 400 iterations for fine-tuning across all testcases. An additional 200 iterations are allocated (i.e., twice the suggested iterations in the *CT* repo [54]) to ensure that *CT* has sufficient opportunity to achieve its best results. Due to significant runtime and resource usage, we cap the extension at 200 iterations. We then report the best placement cost found across all iterations. For training that diverges, we restart the run at least two more times to confirm the divergence.[13]

VI. Experiments and Results

In this section, we first evaluate the performance of *CT* and other macro placers. We then present results of stability and ablation studies. All scripts are public in [63].

A. Comparison of CT with other macro placers

Configuration of different macro placers. We generate macro placement solutions using *CT*, CMP, *SA* and RePlAce. We also include macro placement solutions generated by human-experts. For *CT*, we provide three results: training AlphaChip (i.e., the latest version of the renamed framework in the Circuit Training repository [53]) from scratch (denoted as *CT-Scratch*); fine-tuning AlphaChip using the pre-trained checkpoint released in August 2024 (denoted as *CT-AC*); and fine-tuning AlphaChip using our own checkpoint pre-trained with specific testcase variants (denoted as *CT-Ours* (see Subsection VII-A for details). Due to the very high runtime of pre-training, we run *CT-Ours* only for the Ariane designs. We follow the default settings in [53], except that we use a density weight of 0.5 instead of 1.0, based on guidance from Google engineers [56]. For CMP, we use the default tool settings. For *SA*, we use the configurations described in

[13]One exception: training of *CT-Ariane-X4* from scratch was abandoned after loss and placement return stayed flat in three attempts with 200 iterations.

Section IV and the C++ implementation available in [63]. For RePlAce, we use OpenROAD [49] and set the target density as $util + (1 - util) * 0.5$, where $util$ is the floorplan density of the design. All experiments use Genus 21.1 for synthesis (Fig. 2, Step 1) and Innovus 21.1 for P&R (Steps 5 and 6).

Evaluation of *Nature Table 1* metrics for different macro placers. We generate macro placement solutions for testcases in open NanGate45 (NG45) and ASAP7 and commercial GlobalFoundries 12nm (GF12) enablements. Table III presents *Nature Table 1* metrics obtained using the evaluation flow of Figure 2 for different macro placers on our testcases. The *Table 1* metrics in GF12 are normalized to protect foundry IP: (i) standard-cell area is normalized to core area; (ii) total power and routed wirelength (rWL) are normalized to the *CT-AC* result; and (iii) timing metrics (WNS, TNS) are normalized to the target clock period (TCP) which we leave unspecified.[14] In NG45, the default TCP values for Ariane, BlackParrot Quad-Core (BlackParrot), and MemPoolGroup (MemPool) are 1.3ns, 1.3ns, and 4.0ns. In ASAP7, they are 0.9ns, 0.85ns, and 1.8ns. All testcases reported in Table III have 68% floorplan utilization, matching the Ariane design that is public in *CT*.

Table III also reports *CT* proxy cost for all macro placement solutions, as evaluated by *CT*'s *plc_client*. To compute proxy cost for CMP, RePlAce, *SA* and human-expert solutions, we first update hard macro locations and orientations, then run FD placement via the *plc_client* to place all standard-cell clusters (soft macros). We then compute the proxy cost. Figure 3 shows Ariane-ASAP7 macro placements produced by the macro placers we study.[15] We observe the following.

- **Comparison of routed wirelength (rWL):** CMP consistently dominates the other macro placers except on Ariane-NG45, where CMP's rWL is 2% worse than *SA*'s. For BlackParrot-GF12, CMP has 34% less rWL than *CT-AC*.
- **Comparison of proxy cost:** *SA* dominates other macro placers in 6 of 9 cases, *CT-AC* dominates other macro placers in 2 of 9 cases, and *CT-Scratch* dominates other macro placers in 1 of 9 cases.
- **Comparison between *CT-AC* and *CT-Ours*:** *CT-AC* consistently outperforms *CT-Ours* in proxy cost for the Ariane testcase in all three technologies (NG45, ASAP7, GF12).
- **Comparison between *CT-AC* and *SA*:** *CT-AC* yields better TNS than *SA* for 6 of 9 cases, while *SA* gives better rWL (7 of 9 cases) and proxy cost (6 of 9 cases) than *CT-AC*.
- **Comparison between *CT-AC* and Human experts:** For large macro-heavy designs such as BlackParrot and MemPoolGroup, human experts dominate *CT-AC* in terms of the *Nature Table 1* metrics of postRouteOpt "ground truth" [29] outcomes (5 of 6 cases). For these large designs, even though the proxy congestion cost for *CT-AC* is better than that of the human expert, both macro placement solutions finish routing DRC-clean or with very similar DRCs in ASAP7, as mentioned in Footnote 15.
- **Comparison of resource usage:** Resource requirements vary across the macro placers. Considering that 1 GPU is equivalent to 10 CPUs, i.e., following the analysis in [29], resources ($\#cpus \times runtime$) used for BlackParrot-NG45 by *CT-Scratch*, *CT-AC*, *SA*, RePlAce, and CMP are, respectively, $(8 \times 10 + 256) \times 56.76 = 19,068$, $(8 \times 10 + 256) \times 64.01 = 21,507$, $80 \times 11.2 = 896$, $1 \times 0.31 = 0.31$, and $8 \times 0.33 = 2.64$ CPU-hours.[16] Human expert solutions are generated in under 12 hours for each design.
- ***SA* runtime calibration:** We report *SA* runtimes on an Intel Xeon Gold 6148 CPU. However, we have observed that for some testcases, runtime can be as low as one-third of this reported runtime when using an AMD EPYC 7742 CPU.
- **Comparison of *SA* with *SA* in [6]:** Compared to the *SA* in [6], our updated *SA* with GWTW achieves better proxy cost in two of the three NG45 testcases (geometric-mean improvement of 9% and up to 26%), better routed wirelength and total power in all three cases, and better TNS in one of the three cases. Note that *SA* in [6] finds better proxy values than *CT-AC* for two of the three NG45 testcases.

Evaluation of different macro placers on *CT-Ariane* and its scaled versions. Table IV presents results for Google's public TSMC 7nm Ariane testcase (*CT-Ariane*) and its scaled (x2, x4) versions. Since the LEF/DEF files generated from protobuf [53], do not include the timing or layout information required for a complete P&R flow, we only report post-detailed placement half-perimeter wirelength (DP-HPWL) along with proxy cost.[17] We observe the following.

- Both *CT-AC* and *CT-Scratch* produce better proxy cost (in all three components) than the reference (from Google's internal runs) provided in the *CT* repo [53].[18] This indicates that we are training *CT-AC* and *CT-Scratch* correctly.
- *SA* dominates both *CT-AC* and *CT-Scratch* in terms of DP-HPWL and proxy cost, while using only a fraction of the runtime and compute resources.
- We follow the "quantified scaling suboptimality" methodology in [17] to perform scaling studies of different macro placers. We define the *scaling suboptimality* $\alpha(k)$ as

$$\alpha(k) = \frac{\text{DP-HPWL\_}k}{(k \times \text{DP-HPWL\_1})} - 1 \qquad (1)$$

where DP-HPWL_k is the DP-HPWL of the scaled (kx) version of the base design. A lower value of $\alpha(k)$ indicates better scaling behavior. For *CT-AC*, *SA*, RePlAce and Human solutions, ($\alpha(2)$, $\alpha(4)$) are (0.000, 0.037), (0.009, 0.064), (0.005, -0.022) and (0.003, 0.028), respectively.

[14]WNS and TNS timing metrics in Table III suggest that TCP for MemPoolGroup-GF12 could be increased; we report such improvements and updates in [64].

[15]All postRouteOpt designs are DRC-clean, except that in our ASAP7 enablement, we observe spacing violations across all macro placement solutions around macro blockages for MemPool Group design. These violations arise from a few macro pins being covered by the blockage layer, resulting in total DRC counts below 200.

[16]Or, in financial terms, running the *CT* model on Google Cloud [45] with one 8 NVIDIA V100 GPU train server and five collect servers costs approximately $\$(17.01+5\times3.21) = \$33.06$/hr, while *SA* runs cost \$3.21/hr.

[17]We run the *place_design* command in Innovus 21.1 to place standard cells, then report "Total half perimeter of net bounding box" (DP-HPWL) from the log of the 'earlyGlobalRoute' command.

[18]We use for comparison the best result (run_07) reported in [53].

TABLE III

PPA, PROXY COST, RUNTIME, AND RESOURCE DETAILS OF DIFFERENT MACRO PLACEMENT SOLUTIONS ON OUR MODERN BENCHMARKS, ACROSS THREE DIFFERENT ENABLEMENTS. GF12-BASED RESULTS ARE NORMALIZED. † INDICATES THAT A TOTAL OF 400 ITERATIONS ARE USED FOR TRAINING. BEST RWL, TNS AND PROXY COST VALUES FOR EACH DESIGN, ACROSS ALL MACRO PLACEMENT METHODS, ARE HIGHLIGHTED USING BLUE BOLD FONT. #G AND #C DENOTE THE NUMBER OF V100 GPUS AND CPU THREADS, RESPECTIVELY.

Design Tech	Macro Placer	PostRouteOpt PPA (From Innovus)					Proxy Cost Details				Runtime (Hrs) (#G, #C)
		Area (μm^2)	rWL (μm)	Power (mW)	WNS (ps)	TNS (ns)	WL	Den.	Cong.	Proxy	
Ariane NG45	CT-Scratch†	246303	4648156	832	-140	-119.1	0.102	0.518	0.973	0.848	36.26 (8, 576)
	CT-AC†	248382	4995968	836	-88	-52.2	0.101	**0.508**	0.931	0.820	36.18 (8, 576)
	CT-Ours†	245703	4898125	831	-86	-57.8	0.108	0.538	0.966	0.860	38.79 (8, 576)
	SA	247777	**3976569**	827	-126	-116.8	0.090	0.515	**0.907**	**0.801**	11.52 (0, 80)
	RePlAce	251117	5131963	842	-99	-94.0	0.092	0.998	1.748	1.465	0.04 (0, 1)
	CMP	256230	4057140	852	-154	-196.5	**0.088**	0.909	1.455	1.270	0.05 (0, 8)
	Human	249034	4681178	832	-88	**-46.8**	0.107	0.738	1.376	1.164	NA
BlackParrot NG45	CT-Scratch†	1990312	47785761	4822	-205	-1203.4	0.096	0.790	1.132	1.057	56.76 (8, 576)
	CT-AC†	1944272	33165774	4569	-230	-1486.5	0.066	0.755	1.053	0.970	64.01 (8, 576)
	SA	1938779	28937792	4512	-230	-3012.8	0.054	**0.711**	**0.936**	**0.878**	11.20 (0, 80)
	RePlAce	1930960	26854143	4485	-191	-868.0	0.050	1.049	1.153	1.151	0.31 (0, 1)
	CMP	1916166	**23144317**	4429	-144	-356.2	**0.050**	0.882	1.066	1.024	0.33 (0, 8)
	Human	1919928	25915520	4470	-97	**-321.9**	0.054	1.158	1.260	1.263	NA
MemPool Group NG45	CT-Scratch†	4915555	119607588	2754	-163	-47.7	0.064	1.200	1.232	1.280	91.77 (8, 576)
	CT-AC†	4871665	112486298	2683	-51	-32.1	0.062	**1.006**	**1.086**	**1.108**	91.39 (8, 576)
	SA	4915598	115229509	2720	-32	-5.9	0.062	1.131	1.095	1.175	11.90 (0, 80)
	RePlAce	4930394	113315081	2688	-96	-7.3	**0.056**	1.621	1.652	1.693	0.88 (0, 1)
	CMP	4837150	**102907484**	2587	-20	**-1.0**	0.057	1.495	1.554	1.581	1.97 (0, 8)
	Human	4873872	107597894	2640	-49	-11.9	0.067	1.586	1.710	1.715	NA
Ariane ASAP7	CT-Scratch†	16570	1026239	505	-142	-184.2	0.119	0.821	0.871	0.965	36.53 (8, 576)
	CT-AC	16524	1014938	505	-108	-105.0	0.122	0.804	0.850	0.950	36.35 (8, 576)
	CT-Ours†	16612	1033863	505	-144	-204.1	0.125	**0.811**	0.873	0.967	39.95 (8, 576)
	SA	16467	886776	503	-124	-141.1	0.108	0.817	**0.822**	**0.928**	10.87 (0, 80)
	RePlAce	16410	917539	504	-108	-124.0	**0.102**	1.169	1.160	1.266	0.02 (0, 1)
	CMP	16350	**843757**	504	-124	-146.1	0.102	1.122	1.141	1.233	0.04 (0, 8)
	Human	16613	1182350	508	-104	**-81.8**	0.131	1.177	1.484	1.461	NA
BlackParrot ASAP7	CT-Scratch†	126524	11380551	1609	-226	-2043.7	0.089	0.908	1.002	1.044	55.87 (8, 576)
	CT-AC†	124987	8880315	1569	-201	-1448.6	0.067	0.848	0.833	0.908	58.27 (8, 576)
	SA	123141	7266869	1547	-120	-424.8	**0.053**	**0.758**	**0.751**	**0.808**	9.78 (0, 80)
	RePlAce	123205	6718623	1540	-96	-590.0	0.064	1.097	1.066	1.145	0.20 (0, 1)
	CMP	122603	**6104230**	1529	-111	**-240.4**	0.058	1.058	0.936	1.055	0.65 (0, 8)
	Human	122914	6521501	1536	-89	-356.6	0.057	1.204	1.053	1.186	NA
MemPool Group ASAP7	CT-Scratch†	Divergence									
	CT-AC†	339535	27208664	1402	-122	-629.0	0.072	**1.170**	**0.812**	**1.063**	112.87 (8, 576)
	SA	338798	26898162	1402	-169	-941.0	0.069	1.305	0.834	1.139	10.19 (0, 80)
	RePlAce	338781	26239567	1387	-152	-819.9	0.063	1.740	1.319	1.593	0.60 (0, 1)
	CMP	338559	**23259139**	1343	-88	-224.9	**0.060**	1.756	1.207	1.541	1.09 (0, 8)
	Human	338457	24573102	1354	-84	**-193.4**	0.073	1.758	1.326	1.614	NA
Ariane GF12	CT-Scratch	0.139	1.018	1.006	-0.128	**-106.6**	0.095	0.529	0.689	0.704	37.42 (8, 576)
	CT-AC	0.139	1.000	1.000	-0.194	-201.3	0.092	0.528	0.678	0.695	37.22 (8, 576)
	CT-Ours†	0.139	1.093	1.016	-0.135	-120.9	0.107	0.554	0.705	0.736	35.32 (8, 576)
	SA	0.138	0.894	0.993	-0.159	-176.2	0.092	**0.522**	**0.677**	**0.692**	10.16 (0, 80)
	RePlAce	0.140	1.016	1.018	-0.168	-197.4	0.093	0.550	0.673	0.704	0.03 (0, 1)
	CMP	0.139	**0.843**	0.993	-0.159	-142.3	**0.082**	0.748	0.831	0.871	0.04 (0, 8)
	Human	0.137	1.037	0.983	-0.139	**-106.6**	0.104	0.914	1.156	1.139	NA
BlackParrot GF12	CT-Scratch†	0.192	1.189	1.025	-0.099	-80.0	0.088	0.861	0.842	0.940	48.10 (8, 576)
	CT-AC†	0.191	1.000	1.000	-0.059	-42.7	0.087	0.754	0.752	0.825	51.10 (8, 576)
	SA	0.190	0.867	0.978	-0.084	-45.5	0.058	**0.610**	**0.640**	**0.683**	8.90 (0, 80)
	RePlAce	0.191	0.751	0.967	-0.098	-143.9	0.056	1.027	0.865	1.002	0.17 (0, 1)
	CMP	0.190	**0.662**	0.949	-0.087	-138.9	**0.051**	0.871	0.779	0.876	0.29 (0, 8)
	Human	0.189	0.709	0.954	-0.049	**-15.4**	0.054	1.152	0.949	1.105	NA
MemPool Group GF12	CT-Scratch†	0.413	1.105	1.074	-0.178	-2201.3	0.074	1.196	0.869	1.107	91.73 (8, 576)
	CT-AC†	0.411	1.000	1.000	-0.171	-2061.6	0.069	**0.960**	0.788	**0.943**	89.83 (8, 576)
	SA	0.408	1.021	0.994	-0.186	**-1499.3**	0.068	1.020	**0.756**	0.956	10.68 (0, 80)
	RePlAce	0.409	0.980	0.982	-0.209	-1858.5	0.059	1.629	1.250	1.499	0.68 (0, 1)
	CMP	0.405	**0.857**	0.918	-0.197	-1961.3	**0.059**	1.526	1.183	1.413	1.19 (0, 8)
	Human	0.406	0.928	0.944	-0.149	-1766.5	0.069	1.523	1.278	1.469	NA

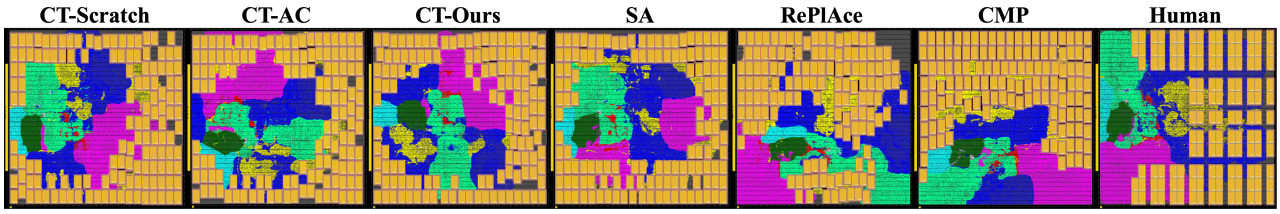


Fig. 3. Macro placement solutions for the Ariane-ASAP7 design, produced by different macro placers.

TABLE IV
DP-HPWL AND PROXY COST, RUNTIME, AND RESOURCE DETAILS OF DIFFERENT MACRO PLACEMENT SOLUTIONS ON *CT-ARIANE* AND ITS SCALED (X2, X4) VERSIONS. † INDICATES A TOTAL OF 400 ITERATIONS USED FOR TRAINING. BEST DP-HPWL AND PROXY COST VALUES ARE HIGHLIGHTED USING BLUE BOLD FONT. #G AND #C DENOTE THE NUMBER OF V100 GPUS AND CPU THREADS, RESPECTIVELY.

Design	Macro Placer	DP-HPWL (μm)	Proxy Cost Details				Runtime (Hrs) (#G, #C)
			WL	Den.	Cong.	Proxy	
CT-Ariane	CT repo [54]	NA	0.098	0.511	0.868	0.787*	NA
	CT-Scratch†	958137	0.094	**0.503**	0.854	0.772	37.89 (8, 576)
	CT-AC†	938528	0.092	0.509	0.829	0.761	55.16 (8, 576)
	SA	804228	**0.081**	0.525	**0.814**	**0.750**	11.13 (0, 80)
	RePlAce	952429	**0.081**	0.992	1.285	1.219	0.03 (0, 1)
	CMP	**745370**	0.075	0.743	0.999	0.946	0.02 (0, 16)
	Human	931105	0.093	0.824	1.241	1.126	NA
CT-Ariane -X2	CT-Scratch	2910809	0.101	0.533	1.015	0.876	38.90 (8, 576)
	CT-AC†	1876365	0.071	0.493	0.836	0.735	86.45 (8, 576)
	SA	1623056	**0.067**	**0.490**	**0.834**	**0.729**	6.93 (0, 80)
	RePlAce	1913954	0.078	0.754	1.091	1.000	0.06 (0, 1)
	CMP	**1510219**	0.074	0.620	1.134	0.951	0.04 (0, 16)
	Human	1868380	0.081	0.832	1.227	1.111	NA
CT-Ariane -X4	CT-Scratch	Divergence					
	CT-AC†	3893091	0.056	**0.466**	0.836	0.707	188.69 (8, 576)
	CT-Ours†	5525222	0.075	0.473	0.922	0.772	174.07 (8, 576)
	SA	3423907	**0.052**	0.467	**0.815**	**0.693**	9.97 (0, 80)
	RePlAce	3726672	0.055	0.730	1.062	0.950	0.17 (0, 1)
	CMP	**3049402**	0.055	0.735	1.139	0.992	0.08 (0, 16)
	Human	3827163	0.060	0.757	1.590	1.233	NA

*We compute the proxy cost for *CT-Ariane* using the cost components reported in the *CT* repo [54], with weights of 1.0 for wirelength and 0.5 for both density and congestion.

B. Stability studies

Outcomes from *CT* training, *SA* execution, and hMETIS-based *grouping* will all vary according to given input seeds.[19] We have studied how this can affect final proxy cost and postRouteOpt PPA metrics. In the following, we present stability issues observed in *CT*, then analyze seed effects in *CT-Scratch*. We then examine seed effects on *SA*, and seed effects on hMETIS-based grouping.

Stability issues of *CT*. We observe that on the same machine, in the same environment, and for the same netlist and seed (e.g., the default "333"), *CT* can converge in one run but diverge in another. Further, all of our *CT* runs use machines with identical configurations, and we also observe different outcomes from otherwise identical runs that are executed on different machines, regardless of netlist size. Figure 4 plots loss and train steps per second for the Ariane-GF12 design, where two identically-configured runs result in convergence (red) and divergence (blue). Although both runs reach a similar speed of around 0.8 steps per second, the loss does not decrease for the diverged run. Given this behavior, if a *CT* run diverges, more runs must be launched to confirm divergence. Determining exact causes and resource-efficient mitigations of this stochasticity is an open direction for future investigation.

[19]Perturbing the design (e.g., changing the SDC clock period by ± 1 ps or the floorplan width by ± 1 site) produces different outputs from CMP and RePlAce, thereby modifying the input design. We evaluate seed-induced variation in *SA* and *CT* while holding the input design fixed. Because CMP and RePlAce do not expose a seed and produce deterministic outputs for a fixed input design, we do not perform variation studies for them.

TABLE V
VARIATION IN POSTROUTEOPT PPA AND PROXY COST FOR *CT-SCRATCH* ON ARIANE-ASAP7 ACROSS 9 RUNS (3 GLOBAL SEEDS × 3 RUNS PER SEED).

Seed	PostRouteOpt PPA (From Innovus)					Proxy Cost Details			
	Area (μm^2)	rWL (μm)	Power (mW)	WNS (ps)	TNS (ns)	WL	Den.	Cong.	Proxy
111	16528	1022767	505.6	-128	-133.6	0.123	0.802	0.848	0.947
	16419	991490	504.1	-97	-59.9	0.118	0.809	0.822	0.934
	16410	993215	504.6	-122	-123.8	0.121	0.809	0.842	0.946
222	16446	1014526	502.0	-112	-84.6	0.120	0.807	0.834	0.940
	16537	985154	505.7	-142	-163.4	0.119	0.825	0.828	0.945
	16482	1006802	505.3	-131	-127.4	0.119	0.805	0.843	0.943
333	16528	1013108	504.5	-128	-164.1	0.119	0.821	0.871	0.965
	16630	1016008	506.4	-162	-228.0	0.126	0.801	0.855	0.953
	16602	1026772	506.1	-198	-355.0	0.125	0.829	0.851	0.964
Mean	16509	1007760	504.9	-136	-160.0	0.121	0.812	0.844	0.949
STD	76.8	14651.3	1.34	29.5	87.65	0.003	0.010	0.015	0.010

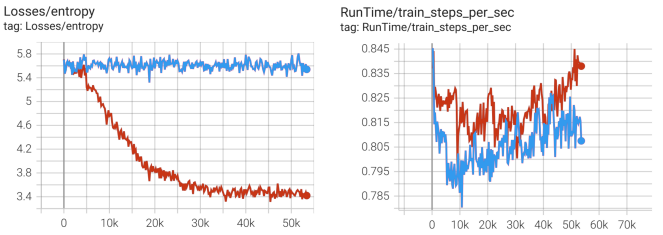


Fig. 4. Loss (left) and train steps per second (right) for converged (red) and diverged (blue) *CT* runs on Ariane-GF12. Both runs use the same machine, same environment, same netlist, and same (default) seed = 333.

Seed effects in *CT*. Despite non-determinism of *CT* training and its outcomes (even with the same seed, environment, and machine), we have sought to understand the variability of *CT* outcomes, as follows. (i) We set three global seeds (111, 222, and 333) for random weight initialization. (ii) We then perform three independent runs of *CT-Scratch* for 400 iterations on the Ariane-ASAP7 design *per each seed*, resulting in a total of nine runs. Table V shows final proxy cost and postRouteOpt PPA of these nine runs.[20]

Seed effects in *SA*. *SA* runs are deterministic: when run with the same seed, on the same machine and in the same environment, *SA* produces the exact same result. Table VI gives details of proxy cost and postRouteOpt metrics for the Ariane-ASAP7 design when *SA* is run with ten different seeds. From Tables V and VI, we observe the following.

[20]We note that *Nature* Extended Data Table 5 [29] and [47] discuss seed effects on variation, but not the stochasticity effects that we document here.

TABLE VI
VARIATION IN POSTROUTEOPT PPA AND PROXY COST FOR *SA* ON THE ARIANE-ASAP7 DESIGN ACROSS 10 SEEDS.

Seed	PostRouteOpt PPA (From Innovus)					Proxy Cost Details			
	Area (μm^2)	rWL (μm)	Power (mW)	WNS (ps)	TNS (ns)	WL	Den.	Cong.	Proxy
1	16467	886776	503.5	-124	-141.1	0.108	0.817	0.822	0.928
111	16503	902468	503.6	-120	-131.3	0.111	0.820	0.845	0.943
222	16380	901541	500.7	-89	-62.1	0.108	0.817	0.802	0.917
333	16494	896866	504.7	-102	-83.7	0.108	0.803	0.848	0.934
444	16401	900137	503.9	-70	-37.3	0.108	0.818	0.812	0.923
555	16461	894386	504.1	-81	-56.9	0.105	0.814	0.813	0.919
666	16420	898643	504.3	-109	-95.7	0.107	0.802	0.808	0.912
777	16563	898613	504.3	-118	-113.1	0.107	0.815	0.835	0.933
888	16486	898399	504.8	-133	-110.3	0.106	0.822	0.806	0.920
999	16522	904774	502.6	-121	-105.3	0.107	0.813	0.810	0.918
Mean	16470	898260	503.7	-105	-93.7	0.107	0.815	0.822	0.926
STD	56.4	4984.1	1.21	20.6	33.47	0.002	0.007	0.017	0.010

- For proxy cost, *CT-Scratch* and *SA* show very similar variation, which differs from the observation in [6]. Decreased variation of proxy cost for *CT-Scratch* may be due in part to updates in the new version of *CT* and to our training for 400 iterations, as opposed to 200 in [6]. Increased variation in *SA* proxy cost may be due to use of 80 *SA* workers, as opposed to 320 workers in [6].
- For PPA (rWL, power and TNS), *SA* shows significantly less variation than *CT-Scratch*. Furthermore, mean PPA and proxy cost values from *SA* are better than those from *CT-Scratch*, indicating that *SA* outperforms *CT-Scratch*.

Seed effects in CT-Grouping. The CT-Grouping flow in *CT* uses the *hMETIS* binary, which does not support a seed input and is non-deterministic. An example consequence is that we cannot reproduce the generation of the clustered netlists used in [6]. However, the *hMETIS* shared library C API does support specification of a seed: we have written a C++ wrapper (publicly available in [63]) which accepts a seed and ensures reproducibility of the grouping flow. To study the effect of seeds in grouping, we generate five clustered netlists for Ariane-ASAP7 using seeds $(111, 222, \ldots, 555)$; these netlists respectively have 782, 791, 784, 785 and 786 standard-cell clusters. We then run *SA* and evaluate the macro placement solutions using the evaluation flow described in Subsection V-B. Table VII presents the postRouteOpt PPA and proxy cost for these five runs. We see that variation in proxy cost is similar to that seen in the seed study of *SA* (Table VI), while there is larger variation in postRouteOpt PPA metrics.

Further, we study the *combined* effects from seeding of CT-Grouping and seeding of *SA*, by "crossing" the first six rows of Table VI with the five rows of Table VII. That is, for each of the five clustered netlists generated using different grouping seeds, we run six *SA* runs with different seeds, and capture resulting postRouteOpt PPA and proxy costs. Mean (standard deviation, STD) for power, WNS, TNS and proxy cost are respectively 504.3 (0.69), -117 (59.1), -106.5 (52.27) and 0.929 (0.009). For proxy cost, the combined effect of *SA* and grouping is similar to the effect of only grouping or only *SA*. For PPA, the combined effect shows less variation in power, WNS and TNS compared to varying only the grouping seed. Compared to varying only the *SA* seed, variation in TNS and WNS increases while variation in power decreases.

TABLE VII
VARIATION IN POSTROUTEOPT PPA AND PROXY COST FOR *SA* ON 5 DIFFERENT CLUSTERED NETLISTS OF THE ARIANE-ASAP7 DESIGN, WHEN HMETIS IS RUN WITH 5 DIFFERENT SEEDS IN THE *CT* GROUPING FLOW.

GRP Seed	PostRouteOpt PPA (From Innovus)					Proxy Cost Details			
	Area (μm^2)	rWL (μm)	Power (mW)	WNS (ps)	TNS (ns)	WL	Den.	Cong.	Proxy
111	16412	896408	502.6	-97	-74.8	0.104	0.830	0.806	0.922
222	16434	886847	504.3	-104	-72.7	0.107	0.815	0.817	0.923
333	16448	907544	503.6	-95	-91.8	0.104	0.824	0.802	0.917
444	16431	915978	504.4	-104	-98.5	0.104	0.843	0.790	0.920
555	16433	904477	503.9	-398	-219.7	0.109	0.850	0.806	0.937
Mean	16432	902251	503.7	-160	-111.5	0.106	0.832	0.804	0.924
STD	12.8	11100.2	0.70	133.3	61.47	0.002	0.014	0.010	0.008

TABLE VIII
EFFECT OF INITIAL PLACEMENT ON *CT-SCRATCH* OUTCOME FOR ARIANE-ASAP7.

Initial Placement	PostRouteOpt PPA (From Innovus)					Proxy Cost Details			
	Area (μm^2)	rWL (μm)	Power (mW)	WNS (ps)	TNS (ns)	WL	Den.	Cong.	Proxy
Genus iSpatial	16570	1026239	505.4	-142	-184.2	0.119	0.821	0.871	0.965
Center	16444	984576	504.3	-108	-79.0	0.119	0.789	0.840	0.933
Lower Left	16466	951872	503.6	-121	-113.2	0.110	0.795	0.834	0.925
Upper Right	16416	992912	504.7	-105	-83.0	0.115	0.821	0.829	0.939
No Placement	16399	975230	502.3	-98	-81.9	0.129	0.843	0.870	0.986

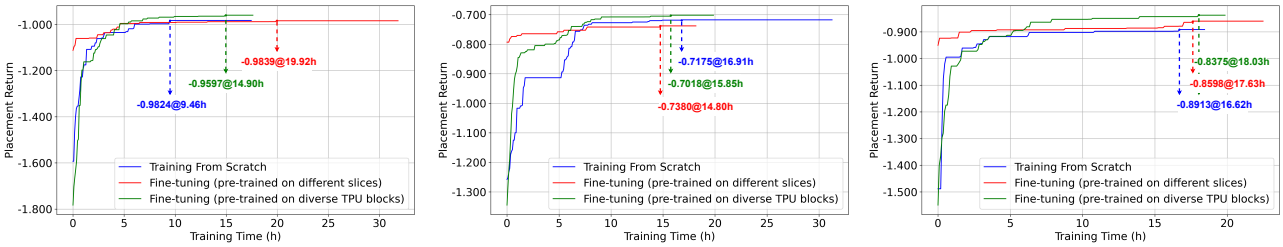
C. Ablation studies

We present results and takeaways from ablation, stability and related studies; further examples appear in [63] [64].

Effect of initial placement on the *CT* outcome. *CT*'s use of a physical synthesis tool (Synopsys DC-Topographical) and the initial placement locations that it outputs with the gate-level netlist is well-discussed in [47] [14]. To evaluate the effect of initial placement, we generate four new clustered netlists for the Ariane-ASAP7 testcase when all cells are placed (i) at the lower left corner (0, 0), (ii) in the middle of the canvas (111.204, 110.970), (iii) at the upper right corner (222.408, 221.940), and (iv) with no placement information.[21] We run CT from scratch for 400 iterations to obtain a macro placement for each clustered netlist, then run our evaluation flow to capture postRouteOpt PPA. Table VIII shows that PPA numbers are very similar to those from the default run – which uses placement information from Genus iSpatial. This observation differs from that of [6], which found up to 10% improvement in rWL when a Genus iSpatial placement solution was used. ([6] used the older version of CT, trained for 200 iterations, and tested on the Ariane-NG45 design.)

Correlation of proxy cost to *Nature Table 1* metrics. The RL agent in *Nature* and *CT* is driven by proxy cost, while the EDA tool's post-P&R output is "ground truth" [29]. To study correlation of proxy cost with *Nature Table 1* metrics, we collect 30 macro placements produced by *CT-Scratch* for Ariane-ASAP7 having proxy cost less than 1.0, and generate *Table 1* metrics for each. Table IX shows the Kendall rank correlation of proxy cost and *Table 1* metrics. Values close to +1 (resp. -1) indicate strong correlation (resp. anticorrelation), while values close to 0 indicate lack of correlation. Similar to [6], in the regime of relatively low proxy cost (high solution quality), we observe poor correlation of proxy cost and its components with *Table 1* metrics. Compared to the proxy cost correlation study on Ariane-NG45 in [6] (Table 2), we see

[21] To model the absence of placement information, we turn off the *breakup* flag in the CT grouping flow.



(a) (b) (c)

Fig. 5. The impact of pre-training versus training from scratch on performance (higher placement return on the y-axis is better) for (a) Ariane-ASAP7, (b) Ariane-GF12 and (c) Ariane-NG45. Here, the placement return is defined as the negative of the proxy cost.

TABLE IX
KENDALL RANK CORRELATION COEFFICIENT BETWEEN PROXY COST AND *NATURE TABLE 1* METRICS FOR ARIANE-ASAP7.

Proxy Cost Element	Area	rWL	Power	WNS	TNS
Wirelength	0.046	0.355	0.222	-0.002	0.019
Congestion	0.053	0.297	0.251	0.116	0.149
Density	-0.055	0.299	0.299	-0.039	0.023
Proxy	0.058	0.402	0.320	0.051	0.090

TABLE X
MACROPLACEMENT SOLUTIONS GENERATED FOR ARIANE-ASAP7 USING CMP IN INNOVUS VERSIONS 19.1, 20.1, AND 21.1.

Macro Placer	PostRouteOpt PPA (From Innovus)					Proxy Cost Details			
	Area (μm^2)	rWL (μm)	Power (mW)	WNS (ps)	TNS (ns)	WL	Den.	Cong.	Proxy
CMP 19.1	16407	879781	504	-131	-128.6	0.104	1.256	1.308	1.386
CMP 20.1	16423	884527	504	-155	-196.8	0.103	1.295	1.333	1.417
CMP 21.1	16350	843757	504	-124	-146.1	0.102	1.122	1.141	1.233

that the new *CT-Scratch* on Ariane-ASAP7 shows improved correlation between proxy cost and rWL, but degraded correlation between proxy cost and WNS. Overall, the proxy cost function optimized in the *Nature* paper shows poor correlation with final chip metrics; therefore, relying on this proxy as an optimization target is not a sound physical design strategy.

Effect of different Innovus versions. We have studied whether differences between CMP versions affect our conclusions. We run CMP in Cadence Innovus versions 19.1, 20.1, and 21.1 to obtain three macro placement solutions for the Ariane-ASAP7 design. We then run the evaluation flow using Innovus 21.1 (see Figure 2). Table X gives postRouteOpt PPA metrics for the three macro placement solutions, which are qualitatively very similar. We observe that CMP 21.1 produces the best rWL result, while CMP 19.1 produces the best TNS result.

VII. STUDIES OF PRE-TRAINING

In [6], we do not address pre-training because CT did not provide pre-training scripts. [38] and [54] show that training from scratch achieves results similar to pre-training. The *Nature* paper does not study the impact of pre-training on PPA metrics. However, several recent writings [14] [13] [48] highlight the lack of pre-training in [6]. Here, Google's recent open-sourcing of a recipe for pre-training [53] as well as pre-trained AlphaChip model weights (*CT-AC*) [53] enable us to study pre-training in detail. We are guided by [38] [53], where *Nature* authors describe two ways to leverage pre-training so as to improve fine-tuning results for a given target netlist. (i) The first way is to use Google's published AlphaChip checkpoint, which is pre-trained on 20 diverse TPU blocks [53]; we refer to this as *CT-AC*. (ii) The second way is to pre-train a model using similar slices of the target block, following guidance from [38] [53]; we refer to this as *CT-Ours*.

In the following, the first subsection studies pre-training on variants of target slices, and fine-tuning on the original target slice. This aims to validate "pretraining on different slices of the same block" from [38]. The second subsection studies pre-training on variants of slices of the target block itself, and fine-tuning on the entire block, to help with convergence. This aims to use the pre-trained model to improve CT scalability on the previously diverged large testcase, *CT-Ariane-X4*. The third subsection studies sensitivity to netlist diversity, as well as inherent scalability, of pre-training. Here, our studies show limitations of the pre-training recipe published in [53].

A. Pre-training and fine-tuning on slices

We study pre-training with diverse variants, using the Ariane testcase in three technologies: ASAP7, GF12 and NG45. The variants are generated using a perturbation strategy adapted from [38] (cf. Section 3.2.2 in [38]), where "we pre-trained a model on the first 7 slices, and fine-tune on the 8^{th} slice". In our experiment, all slices have the same internal and I/O logic, but differ in their I/O port locations. We apply three operators to produce variant slices for a given target block: (i) *X-flip* (flip along the x-axis); (ii) *Y-flip* (flip along the y-axis); and (iii) *Shift* (move each I/O clockwise on the canvas boundary, by a distance equal to 3% of the length of the canvas side on which the I/O is located).

Given a target netlist with placed I/O ports, we produce the 7 slices of the pre-training dataset using (i) Shift; (ii) X-flip; (iii) Y-flip; (iv) XY-flip (flip along both axes); (v) Shift-X-flip (X-Flip followed by Shift); (vi) Shift-Y-flip (Y-Flip followed by Shift); and (vii) Shift-XY-flip (XY-Flip followed by Shift). Then, (viii) the 8^{th} slice is the (unchanged) target itself.

For each of the Ariane-ASAP7, Ariane-GF12 and Ariane-NG45 testcases (each having 133 macros), we pre-train a model using the first 7 slices. As recommended by [38] [53], we use 252 collect jobs (36 collect jobs for each slice) to pre-train the model for 200 iterations. We then run three experiments with the 8^{th} slice (i.e., the original target without any change): (i) training *CT* from scratch; (ii) fine-tuning the *CT-Ours* model that we pre-trained on the first 7 slices; and (iii) fine-tuning Google's public *CT-AC* model.

Figure 5 illustrates the performance of these three models – in terms of placement return versus training time – for each testcase. The model pre-trained from scratch takes similar time to converge to a high-quality solution as do the pre-trained models. This is qualitatively different from Figure 8 in [38], where "starting from scratch takes 5x longer to reach a high-

quality placement". Also, the model fine-tuned using *CT-Ours* converges to a worse placement compared to the model fine-tuned using *CT-AC*; this aligns with the observation in [38].

B. Pre-training on slices and fine-tuning on the whole block

As shown in Table IV, *CT-Ariane-X4* diverges when training from scratch. We now explore if pre-training on *CT-Ariane* and *CT-Ariane-X2* variants helps CT-Ariane-X4 converge during fine-tuning. Because *CT-Ariane-X4* can be constructed by replicating *CT-Ariane* four times or *CT-Ariane-X2* twice, we treat *CT-Ariane* and *CT-Ariane-X2* as slices of *CT-Ariane-X4*, again following guidance from [38]. Then, we generate six variant slices – CT-Ariane-X1-{Shift, X-flip, Y-flip}, CT-Ariane-X2-{Shift, X-flip, Y-flip} – for pre-training. With these six slices, we pre-train a model using 252 collect jobs (42 collect jobs per slice) for 200 iterations.

We run three experiments with the original *CT-Ariane-X4*: (i) training *CT* from scratch; (ii) fine-tuning the *CT-Ours* that we pre-trained on six *CT-Ariane* and *CT-Ariane-X2* slices; and (iii) fine-tuning Google's public *CT-AC* model. Each fine-tuning experiment is run for 400 iterations.

Figure 6 shows placement return versus training time for our study. We observe the following. (i) Pre-training enables AlphaChip to converge on a larger netlist for which from-scratch training fails. (ii) The *CT-AC* model pre-trained on 20 diverse TPU blocks ultimately achieves a higher placement return than the *CT-Ours* model pre-trained on other slices; this is consistent with the observation in Subsection VII-A. (iii) The model pre-trained on other slices sees a sharp placement return increase at 50 hours but improves by only 4% further over the next 125 hours. (iv) For *CT-Ariane-X4*, the *CT-AC* model pre-trained on diverse TPU blocks has a lower starting point, unlike what we observe in Figure 5.

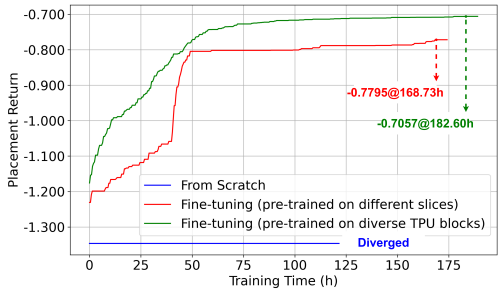


Fig. 6. Placement returns achieved during training from scratch and during fine-tuning from two pre-training strategies: one using data from slices (*CT-Ariane* and *CT-Ariane-X2*), and the other using a diverse set of TPU blocks. Although the model pre-trained on slices produces a sharp increase in return at 50 hours, it does not outperform the model pre-trained on a diverse set of TPU blocks. Here, the placement return is the negative of the proxy cost.

C. Convergence of pre-training

We study how pre-training convergence relates to the diversity of the pre-training dataset and to the size (i.e., #macros) of the largest slice in the pre-training dataset. Our results show limitations of the current pre-training recipe provided in [53].

Diversity. We use a larger testcase, MemPoolGroup, to study the impact of pre-training dataset diversity. We generate seven slices for MemPoolGroup as described in Subsection VII-A. From these seven slices, we create five pre-training datasets that are respectively comprised of 7, 5, 5, 4 and 3 distinct slices. These pre-training datasets are summarized as Set IDs 1-5 in Table XI. For each pre-training dataset, we pre-train a model with at least 252 collect jobs (distributed across slices) for 50 iterations. The rightmost column of Table XI indicates that higher diversity leads to divergence.[22]

Scalability. To investigate the effect of the number of macros, we use the unperturbed netlists of *CT-Ariane*, *CT-Ariane-X2*, and *CT-Ariane-X4* as individual pre-training sets, each containing a single netlist. We again use 252 collect jobs, and all jobs collect samples on the same unperturbed netlist. The last three rows of Table XI show that pre-training on smaller netlists (*CT-Ariane* and *CT-Ariane-X2*) converges, while pre-training on *CT-Ariane-X4* diverges.

Our diversity and scalability studies of pre-training motivate several open questions for future investigation, e.g.:

- How much additional computational resource, and/or additional iterations, are needed to ensure convergence of pre-training with increased diversity of the pre-training dataset?
- When generating slices [38], by how much should each perturbation differ from the original (target) netlist? (Can too-small or too-large perturbations harm convergence?)

TABLE XI
CONVERGENCE OF PRE-TRAINING ACCORDING TO PRE-TRAINING SET DIVERSITY AND NUMBER OF MACROS.

Design	Set ID	Pre-Training Dataset	Max #Macro	#Collect Jobs	Converge?
MemPool Group NG45	1	Shift, {X,Y,XY}-flip, Shift-{X,Y,XY}-flip	324	252	No
	2	Shift, XY-flip, Shift-{X,Y,XY}-flip	324	255	No
	3	Shift, {X,Y}-flip, Shift-{X,Y}-flip	324	255	No
	4	Shift, {X,Y}-flip, Shift-XY-flip	324	256	Yes
	5	Shift, {X,Y}-flip	324	252	Yes
CT-Ariane	6	Original (no change)	133	252	Yes
CT-Ariane-X2	7	Original (no change)	266	252	Yes
CT-Ariane-X4	8	Original (no change)	532	252	No

VIII. Conclusions

Google's *Nature* paper [29], and the subsequent GitHub releases of Circuit Training and its "AlphaChip" update [53], have drawn broad attention throughout the EDA and IC design communities. To the best of our knowledge, no successful reproduction by others of claims in [29] has been published in conferences or journals as of November 2025. Meanwhile, *Nature* authors have made updates to *CT* during the 2.5 years between the commits studied in [6] and in this work; notably, these include much-welcomed pre-training recipes and the pre-trained AlphaChip model weights. This has motivated our continued efforts toward open, transparent implementation and a more rigorous assessment of *Nature* and *CT*.

In this work, we train Google's AlphaChip from scratch and fine-tune AlphaChip (from the pre-trained checkpoint released in August 2024), for all our testcases. We strengthen the simulated annealing baseline by incorporating multi-threading and a 1994 "go-with-the-winners" metaheuristic. Importantly, we add sub-10nm experimental enablement: (i) *CT-Ariane* translated from Google's protobuf, along with scaled versions; and (ii) porting of our testcases to the open-source academic ASAP7 PDK. We also perform pre-training of *CT* following instructions in the *CT* repo [53] and guided by [38].

Our updated evaluation reconfirms conclusions of [6]. *SA* and human baselines remain superior to the latest Alpha-

[22] In the same pre-training study performed with MemPoolGroup-GF12, all five pre-training jobs diverge.

Chip, with statistically significant differences in proxy cost and postRouteOpt PPA metrics, using substantially fewer resources. Moreover, studies with *scaled* sub-10nm Ariane variants reveal further weaknesses of [29] – in stability, stochasticity, scalability, and compute and runtime demands. [14] notes that "In any case, AlphaChip has been used in production on blocks with over 500 macros". At the same time, our experimental results for *CT-Ariane-X4* (532 macros, TSMC 7nm) indicate that *SA* achieves better results than AlphaChip in both post-detailed placement HPWL and proxy cost, using a fraction of runtime and computing resources.

We draw conclusions from data in experiments performed since the *Nature* publication. (i) As baselines should use best available prior algorithms, using weak or outdated baselines may lead to misleading conclusions. Careful implementation of strong baselines is crucial for reliable assessments. (ii) It remains an open research question whether classical meta-heuristic methods will continue to stay ahead of data-hungry AI/ML methods such as RL, in large-scale discrete optimizations like macro placement, where "ground truth" is post-P&R PPA [29]. Notably, our data show that the proxy cost optimized in *CT* correlates weakly with final post-route PPA metrics (Table IX), underscoring a fundamental misalignment between the optimization target used in *Nature* [29] and ultimate design objectives. (iii) There is no substitute for open access to data and code. Reproducibility requires both well-documented methodologies and the exact code implementation. (iv) Ideally, a physical design tool should be deterministic, i.e., producing the same result given the same machine and parameter settings. If nondeterminism exists, variations in results must be clearly acknowledged and analyzed. And (v) reproducibility is a cornerstone of scientific research. When proposing a new methodology, authors should strive to facilitate replication through comprehensive documentation, results on public benchmarks, and open-source code implementation.

The difficulty of reproducing the methods and results of [29], and the effort spent on *MacroPlacement*, highlight the importance of "frictionless reproducibility" [11], along with open source code and data releases "upon which others then build" [31], in the academic EDA field and its nexus with AI/ML. Policy changes of EDA vendors [20] since late 2022 are a laudable step forward; they enable us to include Tcl scripts for commercial synthesis, place and route flows in the *MacroPlacement* GitHub. We are also encouraged by recent research-community interactions sparked by, e.g., our scaled *CT-Ariane* testcases and open-sourced experimental enablement. As we wrote in [6]: contributions of benchmarks, design enablements, implementation flows and additional studies to the *MacroPlacement* effort are warmly welcomed.

Acknowledgments

We thank David Junkin, Patrick Haspel, Angela Hwang and their colleagues at Cadence and Synopsys for policy changes that permit our methods and results to be reproducible and sharable in the open, toward advancement of research in the field. We thank many Google engineers, in particular Sergio Guadarrama, Guanhang Wu and Joe Jiang, for clarifying many aspects of Circuit Training, and running their internal *CT* flow with our data. We thank the anonymous reviewers for their diligence, and community members as well as a reviewer for independently replicating our SA results. We thank the San Diego Supercomputer Center for compute resources used in our studies.

References

[1] A. Agnesina et al., "AutoDMP: Automated DREAMPlace-based Macro Placement", *Proc. ISPD*, 2023.

[2] T. Ajayi et al., "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project", *Proc. DAC*, 2019, pp. 76:1-76:4.

[3] D. Aldous and U. Vazirani, "Go With the Winners Algorithms", *Proc. FOCS*, 1994, pp. 492-501.

[4] S. Bae, A. Yazdanbakhsh, M.-C. Kim, S. Chatterjee et al., "Stronger Baselines for Evaluating Deep Reinforcement Learning in Chip Placement", August 2022. https://statmodeling.stat.columbia.edu/wp-content/uploads/2022/05/MLcontra.pdf (Author listing obtained from [27].)

[5] C.-K. Cheng, A. B. Kahng, I. Kang and L. Wang, "RePlAce: Advancing Solution Quality and Routability Validation in Global Placement", *IEEE TCAD* 38(9) (2018), pp. 1717-1730.

[6] C.-K. Cheng, A. B. Kahng, S. Kundu, Y. Wang and Z. Wang, "Assessment of Reinforcement Learning for Macro Placement", *Proc. ISPD*, 2023, pp. 158-166.

[7] C.-K. Cheng, A. B. Kahng, S. Kundu, Y. Wang and Z. Wang, "Assessment of Reinforcement Learning for Macro Placement", 2023, arXiv version, v2. https://arxiv.org/abs/2302.11014

[8] R. Cheng, X. Lyu, Y. Li, J. Ye, J. Hao and J. Yan, "The Policy-Gradient Placement and Generative Routing Neural Networks for Chip Design", *Proc. NeurIPS*, 2022, pp. 26350-26362.

[9] R. Cheng and J. Yan, "On Joint Learning for Solving Placement and Routing in Chip Design", *Proc. NeurIPS*, 2021, pp. 16508-16519.

[10] V. A. Chhabria et al., "Strengthening the Foundations of IC Physical Design and ML EDA Research", *Proc. ICCAD*, 2024.

[11] D. Donoho, "Data Science at the Singularity", *Harvard Data Science Review*, 2024. https://doi.org/10.1162/99608f92.b91339ef

[12] Z. Geng, J. Wang, Z. Liu, S. Xu and Z. Tang, "Reinforcement Learning within Tree Search for Fast Macro Placement", *Proc. ICML*, 2024.

[13] A. Goldie, A. Mirhoseini and J. Dean, "That Chip Has Sailed: A Critique of Unfounded Skepticism Around AI for Chip Design", *arXiv:2411.10053*, 2024.

[14] A. Goldie at al., "Addendum: A graph placement methodology for fast chip design", *Nature* 634, (2024), pp. E10–E11.

[15] H. Gu et al., "LAMPlace: Legalization-Aided Reinforcement Learning-Based Macro Placement for Mixed-Size Designs With Preplaced Blocks", *IEEE TCAS-II* 71(8) (2024), pp. 3770-3774.

[16] J. Gu, H. Gu, K. Liu and Z. Zhu, "An Effective Macro Placement Algorithm Based On Curiosity-driven Reinforcement Learning", *Proc. ISEDA*, 2023, pp. 364-368.

[17] L. Hagen, J. H. Huang and A. B. Kahng, "Quantified Suboptimality of VLSI Layout Heuristics", *Proc. DAC*, 1995, pp. 216-221.

[18] J. Hu et al., "Sensitivity-guided Metaheuristics for Accurate Discrete Gate Sizing", *Proc. ICCAD*, 2012, pp. 233-239.

[19] N. P. Jouppi et al., "TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings", *Proc. ISCA*, 2023.

[20] D. Junkin, "Supporting the Scientific Method for the Next Generation of Innovators", DAC-2022 BoF *Open-Source EDA and Benchmarking Summit*. https://open-source-eda-birds-of-a-feather.github.io/doc/slides/BOAF-Junkin-DAC-Presentation.pdf

[21] A. B. Kahng, R. Varadarajan and Z. Wang, "Hier-RTLMP: A Hierarchical Automatic Macro Placer for Large-scale Complex IP Blocks", *IEEE TCAD* 43(5) (2024), pp. 1552-1565.

[22] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing", *Science* 220(4598) (1983), pp. 671-680.

[23] Y. Lai et al.,"ChiPFormer: Transferable Chip Placement via Offline Decision Transformer", *Proc. ICML*, 2023, pp. 18346-18364.

[24] Y. Lai et al., "Maskplace: Fast Chip Placement via Reinforced Visual Representation Learning", *Proc. NeurIPS*, 2022, pp. 24019-24030.

[25] T. P. Le et al., "Toward Reinforcement Learning–based Rectilinear Macro Placement Under Human Constraints", *Proc. ICCAD*, 2023.

[26] Y. Lin et al., "DREAMPlace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement", *IEEE TCAD* 40(4) (2021), pp. 748-761.

[27] I. L. Markov, "Reevaluating Google's Reinforcement Learning for IC Macro Placement", *Comm. of the ACM* 67(11) (2024), pp.60-71.

[28] I. L. Markov, J. Hu and M. Kim, "Progress and Challenges in VLSI Placement Research", *Proc. IEEE* 103(11) (2015), pp. 1985-2003.
[29] A. Mirhoseini et al., "A Graph Placement Methodology for Fast Chip Design", *Nature* 594 (2021), pp. 207-212.
[30] R. A. K. Richardson, S. S. Hong, J. A. Byrne, T. Stoeger and L. Amaral, "The Entities Enabling Scientific Fraud at Scale are Large, Resilient, and Growing Rapidly", *Proc. Natl. Acad. Sci. USA* 122(32) (2025).
[31] A. Spector and P. Norvig, "Google's Hybrid Approach to Research", *Comm. of the ACM* 55(7) (2018), pp. 34–37.
[32] Z. Tan and Y. Mu, "Hierarchical Reinforcement Learning for Chip-Macro Placement in Integrated Circuit", *Pattern Recognition Letters*, 179(C) (2024), pp. 108-114.
[33] L. Udesky, "'Publish or perish' culture blamed for reproducibility crisis", *Nature*, 2025.
[34] A. Vidal-Obiols, J. Cortadella, J. Petit, M. Galceran-Oms and F. Martorell, "Multi-Level Dataflow-Driven Macro Placement Guided by RTL Structure and Analytical Methods", *IEEE TCAD* 40(12) (2020), pp. 2542-2555.
[35] J. Wang, X. Zhang, Z. Tan and M. Zhu, "RTplace: A Reinforcement Learning-based Macro Placement Method with ResNet and Transformer", *Proc. ICICM*, 2024.
[36] L.-T. Wang, Y.-W. Chang and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*, Morgan Kaufmann, 2009.
[37] W. Yao, Y. Lin and L. Li, "Learning Placement Order for Constructive Floorplanning", *Integration: The VLSI Journal* 100 (2025), pp. 102293.
[38] S. Yu et al., "Scalability and Generalization of Circuit Training for Chip Floorplanning", *Proc. ISPD*, 2022.
[39] D. Zhao, S. Yuan, Y. Sun, S. Tu and L. Xu, "DeepTH: Chip Placement with Deep Reinforcement Learning Using a Three-Head Policy Network", *Proc. DATE*, 2023, pp. 1-2.
[40] Ariane RISC-V CPU Repo. https://github.com/openhwgroup/cva6
[41] ASAP7 PDK and Cell Libraries Repo. https://github.com/The-OpenROAD-Project/asap7
[42] BlackParrot Repo. https://github.com/black-parrot/black-parrot
[43] BSG Black-box SRAM Generator Repo. https://github.com/jjcherry56/bsg_fakeram
[44] FakeRAM2.0 Repo. https://github.com/ABKGroup/FakeRAM2.0
[45] Google Cloud Pricing for Compute Engine (Machine Type: n1-standard-96, Boot Disk: 100GB, Region: Iowa (US-Central1)) on March 14, 2025. https://cloud.google.com/products/calculator
[46] MemPool Repo. https://github.com/pulp-platform/mempool
[47] Peer Review File of "A Graph Placement Methodology for Fast Chip Design". https://static-content.springer.com/esm/art%3A10.1038%2Fs41586-021-03544-w/MediaObjects/41586_2021_3544_MOESM1_ESM.pdf
[48] A. Goldie and A. Mirhoseini, "Statement on Reinforcement Learning for Chip Design", March 24, 2023. https://regmedia.co.uk/2023/03/27/ag_az_statement.pdf
[49] OpenROAD Repo (Aug 30, 2024). https://github.com/The-OpenROAD-Project/OpenROAD, *commit hash: f02a3d4*
[50] RePlAce Repo (Mar 14, 2022). https://github.com/mgwoo/RePlAce, *commit hash: f500065*.
[51] ICCAD04 Mixed-size Placement Benchmarks. http://vlsicad.eecs.umich.edu/BK/ICCAD04bench/
[52] NanGate45 PDK. https://eda.ncsu.edu/freepdk/freepdk45/
[53] Circuit Training: An Open-source Framework for Generating Chip Floorplans with Distributed Deep Reinforcement Learning (Feb 6, 2025). https://github.com/google-research/circuit_training, *commit hash: 4c6fd98*.
[54] Circuit training at scale with Ariane RISC-V (Dec 14, 2023). https://github.com/google-research/circuit_training/blob/main/docs/ARIANE.md, *commit hash: ade6ca2*.
[55] NVDLA Hardware. https://github.com/nvdla/hw/tree/nv_small
[56] G. Wu, Google Brain, *personal communication*, August 2022.
[57] J. Jung, *personal communication*, December 2022.
[58] P. Haspel and A. Hwang, *personal communication*, December 2022.
[59] M. Cavalcante and J. Liu, *personal communication*, December 2022.
[60] J. Jiang, Google Brain, *personal communication*, January 2023.
[61] RePlAce in OpenROAD. https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/gpl
[62] RePlAce version used in CT (release date: January 9, 2020). https://github.com/The-OpenROAD-Project/RePlAce/tree/05ef3ee
[63] MacroPlacement. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/march_updates
[64] "Our Progress: A Chronology". https://github.com/TILOS-AI-Institute/MacroPlacement/tree/march_updates/Docs/OurProgress
[65] Implementation of Proxy Cost Computation. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/CodeElements/Plc_client
[66] Implementation of Simulated Annealing. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/march_updates/CodeElements/SimulatedAnnealingGWTW
[67] Implementation of Force-Directed Placement. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/CodeElements/FDPlacement
[68] A. B. Kahng, "For the Record" and Updates, June 2022 - present. https://github.com/TILOS-AI-Institute/MacroPlacement#ForTheRecord
[69] Impact of Initial Placement on CT Outcome. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/Docs/OurProgress#Question1
[70] MacroPlacement testcases. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/Testcases
[71] MacroPlacement tool flow scripts. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/Flows
[72] Proxy cost correlation to postRouteOpt metrics. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/Docs/OurProgress#Question10
[73] Assumptions for Power Delivery Network. https://github.com/google-research/circuit_training/blob/90fbe0e/circuit_training/environment/placement_util.py#L182-L185
[74] Power Delivery Network Script. https://github.com/TILOS-AI-Institute/MacroPlacement/blob/main/Flows/util/pdn_flow.tcl
[75] Effect of different physical synthesis tool on CT solution. https://github.com/TILOS-AI-Institute/MacroPlacement/tree/main/Docs/OurProgress#Question11
[76] FAQ 5. Was Circuit Training intended by Google to provide the code that was used in the Nature paper?. https://github.com/TILOS-AI-Institute/MacroPlacement?tab=readme-ov-file#faqs.
[77] train_ppo.py, Line 55. https://github.com/google-research/circuit_training/blob/e7b2fcf/docs/ARIANE.md?plain=1#L259
[78] "Replication crisis", *Wikipedia*, https://en.wikipedia.org/wiki/Replication_crisis.

Chung-Kuan Cheng is Distinguished Professor of CSE and Adjunct Professor of ECE at the University of California, San Diego. His research interests include machine learning and design automation for microelectronic circuits. He received the Ph.D. degree in Electrical Engineering and Computer Sciences from the University of California, Berkeley.

Andrew B. Kahng is Distinguished Professor of CSE and ECE at the University of California, San Diego. His interests include IC physical design, the design-manufacturing interface, combinatorial optimization, and AI/ML for EDA and IC design. He received the Ph.D. degree in Computer Science from the University of California, San Diego.

Sayak Kundu is a Ph.D. student in the ECE department at the University of California, San Diego. He received his bachelor's degree in Electronics and Telecommunication Engineering from Jadavpur University, Kolkata, in 2017. His research interests include optimization and machine learning applications in IC physical design flow and methodology.

Yucheng Wang is a Ph.D. student in the Computer Science Department of the University of California at San Diego. He received his bachelor's degree in Computer Science from Purdue University, West Lafayette, in 2021. His research interests include standard-cell layout automation, design technology co-optimization strategies, and graph visualization.

Zhiang Wang received the Ph.D. degree in electrical and computer engineering from the University of California, San Diego in 2024. He is currently a postdoctoral researcher at the University of California, San Diego. His current research interests include physical design, GPU-accelerated EDA and machine learning for EDA.