Adaptive FPGA Placement Optimization via Reinforcement Learning

Kevin E. Murray
Dept. of Electrical & Computer Engineering
University of Toronto, Canada
kmurray@ece.utoronto.ca

Vaughn Betz
Dept. of Electrical & Computer Engineering
University of Toronto, Canada
vaughn@ece.utoronto.ca

***Abstract*—Developing new or improved optimization heuristics for Computer Aided Design (CAD) tools is challenging and time consuming, relying on empirical experimentation and researcher experience. In this work we study how this process can be improved by using Reinforcement Learning (RL) to learn effective and adaptive heuristics. Applying these techniques to Field Programmable Gate Array (FPGA) placement, we show our RL-enhanced algorithm outperforms the standard VTR 8 placer, achieving a better run-time & quality trade-off (up to $2\times$ faster for equivalent quality). RL enables the placer to more efficiently explore the solution space and enables it to dynamically adapt to the specific problem instance being solved. We expect further application of advanced RL methods will improve these results, which motivates further exploration of how RL can be applied in the CAD flow.**

***Index Terms*—Field Programmable Gate Array (FPGA), Electronic Design Automation (EDA), Computer Aided Design (CAD), Machine Learning (ML), Reinforcement Learning (RL)**

I. Introduction

Computer Aided Design (CAD) often requires solving complex large scale optimization problems. The large solution spaces necessitate the use of heuristics to find good quality solutions in reasonable run-times. However, the process of designing these heuristic optimization algorithms is itself challenging and time consuming. The process typically involves two stages:

- first, a CAD researcher develops a new algorithm or algorithm enhancement;
- second, they tune their algorithms' parameters to produce suitable quality and run-time trade-offs.

The tuning process is key to producing good results but is often opaque, relying on the researcher's intuition/experience (e.g. which parameters to expose and how to set them) combined with empirical experimentation. Machine Learning techniques offer the potential to automate and improve this process.

In this work we investigate the use of Reinforcement Learning (RL) to improve a Simulated Annealing (SA) based Field Programmable Gate Array (FPGA) placement algorithm. Unlike previous work looking at CAD parameter auto-tuning (which typically wraps around an existing algorithm) [1], [2], we focus on using Reinforcement Learning to develop policies (which also adapt online) for use *within* the optimization algorithm.

II. Reinforcement Learning

Reinforcement Learning is an approach to machine learning which trains an *Agent*, through experience, to achieve an objective by interacting with its environment. The agent receives feedback about its performance via a numeric *reward*, with the RL agent's goal being to maximize its total reward. RL is a very general problem formulation, allowing RL techniques to be applied to a wide range of problems, including robotics [3], finding good neural network architectures [4], and website recommendations [5]. RL techniques have also been shown to be effective for hard search and decision problems such as Chess, Go [6], and real time strategy games [7].

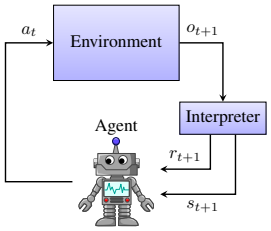

Fig. 1: Reinforcement Learning Problem.

Figure 1 outlines the RL problem in more detail. At timestep t, the agent selects an *action* a_t to perform, from the set of possible actions $\mathcal{A}$. At the next timestep the environment is observed, producing observation o_{t+1}. This observation is then interpreted to produce a *reward* $r_{t+1} \in \mathbb{R}$ and the agent's perception of the environment's *state* s_{t+1} (from the set of possible states $\mathcal{S}$). On the basis of the states and rewards previously observed the agent selects a new action a_{t+1} to perform. For more background on RL see [8].

Many RL techniques rely upon estimating *action values*: the long-term value of taking a particular action (i.e. considering both immediate and future rewards). This estimate is often defined as a function $Q(s, a)$: $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. How Q is estimated from the agent's experience (action value estimation), along with how Q is used to select the next action (action selection) are important considerations for many RL algorithms.

An important special case of the RL problem are so-called K-armed bandit problems. In such problems there is only a single state (i.e. $|\mathcal{S}| = 1$), and Q becomes a function only of the actions: $Q(a)$: $\mathcal{A} \rightarrow \mathbb{R}$.

III. Simulated Annealing Placement for FPGAs

FPGAs are microchips consisting of reconfigurable logic and routing resources, which enable an FPGA to be re-programmed to implement a wide range of digital circuits. As shown in Figure 2, the logic, RAM and other resources are grouped into blocks which are laid out in a grid with routing resources between them.

One of the major steps in the automated FPGA design flow is placement, which determines where to place design logic

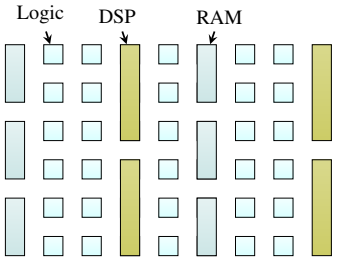

Fig. 2: FPGA placement grid

while minimizing the amount of wiring required and critical path delays. Since design logic can only be placed at locations where pre-fabricated resources exist, FPGA placement is a discrete optimization problem with many legality constraints. It is also key to generating a high quality implementation, as placement is the primary determinant of both routability and timing.[1] These considerations have made Simulated Annealing (SA) a popular algorithm for FPGA placement [9], [10].

As shown in Algorithm 1, SA starts with an initial solution (Line 2). The annealer then repeatedly generates *moves* (Line 5) which perturb the current solution. The associated change in cost (Line 6) is used to decide whether the move should be accepted or rejected. SA accepts all moves which decrease cost, and probabilistically accepts moves which increase cost as a function of the current temperature (Line 7). Accepting cost increases allows the annealer to hill climb and escape local minima. After making M moves (Line 4) the temperature is decreased (Line 9). This focuses the annealer on cost improvements (less likely to accept uphill moves), guiding it to converge to a high quality solution. The process repeats until the exit conditions are met (Line 10).

Algorithm 1 Simulated Annealing Placement

Require: P_{init} initial placement, T_{init} initial temp., M moves per temp.
Returns: An optimized placement
1: **function** SA_PLACE(P_{init}, T_{init}, M)
2: $P \leftarrow P_{init}, T \leftarrow T_{init}$
3: **repeat**
4: **for** $1 \dots M$ **do**
5: $P' \leftarrow$ GENERATE_MOVE(P) ▷ Perturb Solution
6: $\Delta_{cost} \leftarrow$ EVALUATE_MOVE(P, P')
7: **if** PROBABILISTICACCEPT(Δ_{cost}, T) **then**
8: $P \leftarrow P'$ ▷ Accepted Move
9: $T \leftarrow$ UPDATE_TEMP(T)
10: **until** EXIT_CONDITION(P, T)
11: **return** P ▷ Optimized Placement

A. Placement Moves

It is useful to clarify what a move means in the context of FPGA placement. Perhaps the simplest type of move is to swap two blocks of the same type (e.g. two DSP blocks). These are the types of moves used by the standard VTR placer [11], which randomly selects a block from the netlist with uniform probability, and randomly swaps it with another block (or empty location) of the same type. Random swaps are a 'complete' type of move as they ensure any valid placement configuration can be reached from any other. This is desirable since it means potentially better configurations are always reachable through some sequence of moves. Despite this, reaching a significantly better configuration may be challenging, requiring a large number of swaps (potentially many of them uphill if already in a deep local minima).

However moves are not limited to simple random swaps. By exploiting knowledge about the structure of the problem more powerful and intelligent moves can be applied. For instance, *directed moves* [12], [13] use circuit structure, current placement, and timing information to intelligently move blocks towards their optimal positions.[2] These moves make the annealer more effective at escaping local minima (by making larger steps through the solution space), achieving better quality in fewer moves, which reduces run-time.

An interesting and more general way of viewing these more intelligent moves is as a set of algorithms the annealer selects from to improve the current placement. There is no fundamental restriction on what types of algorithms could be used for move generation. Moves could be very significant, potentially encompassing what would traditionally be considered entirely separate placement algorithms, for instance based on partitioning, assignment or analytic placement techniques [14]. In this sense annealing acts as a framework for combining these algorithms together, and as a true meta-heuristic selecting among their results.

However designing such a system which selects among numerous other algorithms is inherently challenging. It is unclear when or how often the different move types should be used. Likely, which type of move will be effective will be situationally dependent, for instance depending on circuit structure and target technology (which varies across designs), the current optimization progress, and the run-time budget. These factors, combined with the large number of possibly diverse move types likely makes it intractable for a human algorithm designer to tune this type of system well. As a result, to control complexity move generators are typically statically configured with limited coupling to the rest of the optimizer.

IV. Reinforcement Learning Enhanced Move Generator

To address this challenge we will apply RL to learn effective policies for controlling more powerful and flexible move generators, aiming to make the optimizer more adaptable. For instance, enabling the optimizer to adapt to both the static characteristics of the problem instance (e.g. circuit structure), and the evolving dynamics of the optimization process (e.g. which types of moves are most productive, which regions of the circuit would benefit from further optimization). This allows the optimizer to focus effort where it is most needed, and explore the solution space more effectively.

An illustration of our RL-enhanced move generator is shown in Figure 3. The 'Agent' chooses between several different types of moves. In this case selecting a different type of block to be randomly swapped. The impact of the move is not known to the agent until after it has been performed.

After selecting a move type, the blocks are swapped and the move evaluated using the existing tool infrastructure to calculate the associated change in cost (Δ_{cost}). The annealer then determines whether the move is accepted or rejected (Algorithm 1 Line 7) as usual. The Δ_{cost} and whether the

[1] Unlike Application Specific Integrated Circuits (ASICs), there are no later stages (e.g. buffer insertion, gate re-sizing) to fix-up timing issues.

[2] For instance, the commercial Quartus placer uses > 10 different types of directed moves [13].

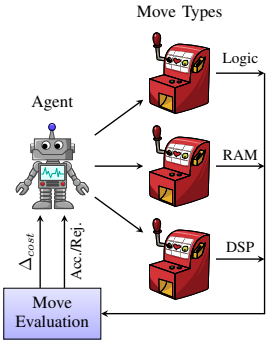

Fig. 3: RL-based move generator.

move was accepted or rejected (i.e. reverted) are provided to the agent to calculate the reward. Based on this feedback, over time the agent learns which move types yield high reward and focuses the placers' effort there.

A. Reward Formulation

To guide the agent towards the desired behaviour we must formulate a reward which captures this intent. The agent will then attempt to maximize this reward through its actions.

One reward formulation we considered was:

$$r_t = -\Delta_{cost} \tag{1}$$

Since placement is a minimization problem (we seek decreases in wirelength and timing costs), but the agent (by convention) seeks to maximize reward, we use the negative of the cost change. Intuitively, this seems to capture our goal: for moves which decrease cost the agent's actions are affirmed (with a positive reward), while for moves which increase cost the agent is penalized (negative reward).

However, we found Equation (1) did not perform well in practise. In particular, the agent prefered move types which had low probability of producing negative rewards. A common example of this is moving IO blocks. Since IO connectivity is not the dominant factor in the placer's cost functions, and given that many IO configurations have similar cost, this allowed the agent to avoid the (potentially) large negative rewards associated with moving other (more significant) block types.

Furthermore, as optimization progresses, it becomes harder to find moves which decrease cost (since the circuit is becoming increasingly well optimized), and correspondingly easier to find moves which increase cost (although such moves become more likely to be rejected as the anneal progresses). This leads the agent to become very risk-averse, focusing on actions with limited chance of a downside – even though it would ultimately be more productive to propose a number of moves with high probability of negative reward (which are likely to be rejected) in the hope of finding some moves which decrease cost.

To better capture this intent we used an alternative reward defined as:

$$r_t = \begin{cases} -\Delta_{cost} & \text{if accepted} \\ 0 & \text{if rejected} \end{cases} \tag{2}$$

This reward formulation does not penalize the agent for proposing moves which are ultimately rejected by the annealer. By reducing the downside the agent becomes more exploratory and focuses on proposing moves which ultimately lead to cost improvements. It is interesting to note this formulation has another advantage: the sum of all rewards is equal to the total cost change of the circuit, which is our ultimate goal.

It is also worth considering how this reward function treats hill climbing moves (which increase cost, but are accepted by the annealer). In this case the agent receives a negative reward which, if a common result, will lead the agent away from proposing that move type. It is unclear whether this is a good choice. Hill climbing moves are helpful for escaping local minima, so penalizing them may not be ideal. On the other hand, it does guide the agent away from unproductive move types, allowing it to focus on more productive types.

B. Estimating Action Values

Our agent's action space is shown in the first row of Table I. Each action corresponds to swapping a random block of a particular type. The FPGA architecture we targeted has four block types, but in general there could be arbitrary types of moves (such as those described in Section III-A). The agent only needs to know the number of potential move types; it does not need to know what they do, or how they work. The agent learns to use them effectively by trying them out and observing the resulting rewards.

As the second row of Table I shows, our agent has only a single state, and hence treats the move type selection problem as a K-armed bandit problem [8]. This is a significant simplifying assumption, and we believe extending the state space will allow the agent to recognize different situations (e.g. optimization progress, previous moves, circuit structure) and respond accordingly. However this is left for future work (Section VI).

One aspect of the move type selection problem which differs from many conventional RL problems is that the problem statistics are *non-stationary*. That is, the distributions of rewards offered by the different move types evolves over time as optimization progresses. For instance, a particular move type may offer the highest expected reward early in placement optimization, but later becomes sub-optimal (i.e. when the placement is better optimized). It is therefore important for our agent to continually adapt to the changing dynamics of the move type rewards.

To estimate the value of taking each potential action, our agent uses a tabular Q function which stores the current estimated reward of taking a particular action a_t. After performing action a_t and receiving reward r_{t+1} the Q value is updated as:

$$Q(a_t) = Q(a_t) + \alpha(r_{t+1} - Q(a_t)) \tag{3}$$

where $r_{t+1} - Q(a_t)$ corresponds to the 'error' between the received reward and the current estimate, and α is the size of step taken to reduce the error.

In order to track the evolving reward statistics we weight the rewards of recent moves more heavily using a weighted exponential average. We correspondingly set α as:

$$\alpha = 1 - e^{\log(\gamma)/M} \tag{4}$$

where M is the number of moves per temperature, and γ is the fraction of weight given to moves which occurred $> M$ moves ago.[3] We empirically found small γ values (0.1 to 0.001)

[3] γ controls the 'length' of the agent's memory. As γ approaches 1 the agent has the longest term memory (Q approaches the average of all previous moves). At $\gamma = 0$ the agent has the shortest term memory (Q equals the reward of the previous move).

TABLE I: Agent State and Action Spaces

Space	Values
Action ($\mathcal{A}$)	{IO, Logic, DSP, RAM}
State ($\mathcal{S}$)	$\{s_0\}$

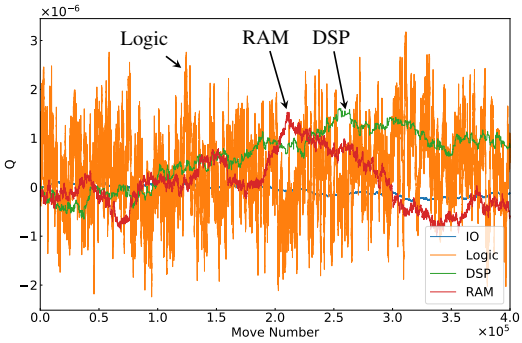

Fig. 4: Agent's perceived action values (Q) during early placement on the `LU8PEEng` benchmark.

performed best as they focus the agent on tracking the shorter term dynamics of which move types were most effective. We expect an expanded state representation (Section VI) will be needed to effectively capture longer term trends.

Figure 4 illustrates how the agent's Q estimates change during early placement with $\gamma = 0.1$.[4] We can see that the Q values for each move type change rapidly as placement proceeds. It is interesting to note the 'best' (highest Q value) move type changes in an oscillatory fashion, alternating between the different block types (Logic, DSP and RAM).

Previous work has found this approach (cyclically optimizing different block types) to be an effective heuristic [15], [16].[5] It is particularly noteworthy that in previous work this heuristic was manually programmed into the placers by their developers. In contrast, we provided no such heuristic guidance to our placer. The agent *learned* this technique by itself, based on its experience and the feedback provided by the reward function. This points toward the potential of using RL to find new improved heuristics and reduce the development effort required to build high quality CAD tools.

C. Action Selection: Exploration versus Exploitation

Once the agent has an estimate of the value of potential actions it must select which action to perform. The agent faces to two competing factors:

- exploring the set of potential actions ($\mathcal{A}$) to improve its action value estimates (Q), and
- performing high pay-off actions to maximize reward.

This is typically referred to in the RL literature as the exploration versus exploitation trade-off [8]. Failing to sufficiently explore possible actions means the agent's value estimates will not reflect the true values – leading to poor choices. However each exploratory action also has a cost, since it is not spent making further progress towards the goal.

$\mathcal{E}$-Greedy action selection is one approach to handle this trade-off. With this technique, the agent mostly selects the highest value (greedy) action – the action it believes will yield maximum reward. However for some fraction of actions ε the agent instead performs a random exploratory action. In stationary RL problems (where the rewards of actions do not evolve) it is common to decrease ε over time, to focus on exploitation, once action value estimates have converged. In our case, the reward statistics are non-stationary so we use a fixed ε .

[4]i.e. placing 90% of weight on the most recent M moves.

[5]Intuitively, after optimizing the placement of a particular block type (e.g. DSPs), it is likely the placement of other block types (e.g. Logic, RAM) could be improved to account for the first type's new placement.

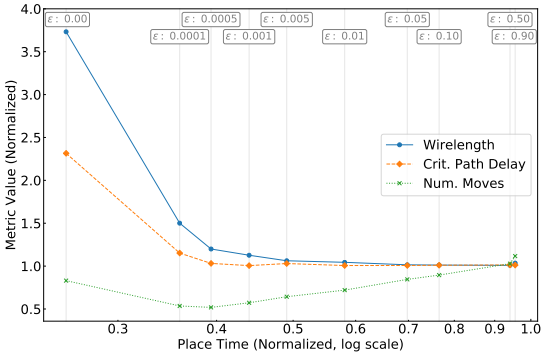

Fig. 5: Run-time trade-off for varying degrees of exploration (ε) with $\gamma = 0.001$, normalized to the default VTR 8 placer. Results are the average over the same benchmark circuits following the methodology of Section V.

Figure 5 shows how varying the fixed ε changes the obtained Quality of Result (QoR) and run-time. First looking at QoR, we observe varying ε has only a small impact impact on Wirelength (WL) and Critical Path Delay (CPD) for most of it's range (0.9 to 0.005).[6] Despite this some exploration is key for good QoR, as decreasing ε (below 0.0005 and particularly to 0.0) significantly degrades QoR. For very small ε the agent performs insufficient exploration (resulting in misleading action value estimates) which leads the agent to myopically focus on sub-optimal move types.

It was surprising such small ε values (e.g. 0.005) did not cause more significant QoR degradation, as they make the agent more exploitative/greedy and hence likely to get stuck in local minima. One reason for this is the agent's relatively short term 'memory' (Section IV-B), which means it tends to forget about historical action performance (whether poor or successful). This ensures the agent continues to adapt, which also makes the agent inherently explorative. Additionally, the move generation process still has a random component, as the locations blocks are moved to are chosen randomly.

Now considering run-time, we also see in Figure 5 that decreasing ε achieves a significant $2.0\times$ to $2.8\times$ run-time reduction for ε from 0.005 to 0.0001. Figure 5 also shows the total number of moves the annealer performs (Num. Moves), which tends to decrease with ε. Making the agent more exploitative (decreasing ε) focuses it on high pay-off move types, which quickly improves the placement in fewer moves. The annealer then exits when it detects no further quality improvement seems likely [9], reducing run-time.

V. Results

We evaluate our RL-enhanced placer (Section IV) on the standard VTR benchmark circuits targeting the `k6_frac_N10_frac_chain_mem32K_40nm` FPGA architecture [11]. We exclude small benchmark circuits with $< 10K$ primitives,

[6]Interestingly, CPD degrades more slowly than WL which is desirable. FPGA designers are typically more concerned with timing than wirelength – so long as their design remains routable.

and average results over three random number generator seeds to reduce algorithmic noise. We compare to the standard VTR 8 placer [11], with two variants of our algorithm using different agents:

- **Random**: selects a block type to move at random (ignoring any learned action values)
- **RL**: selects the block type to move based on action values (exploiting learned action values) using an ε-greedy approach.

Run-time and QoR are both important metrics when evaluating and comparing placement algorithms. Most placement algorithms have some tuneable parameters which allow a trade-off between run-time and quality. This is useful since different run-time/quality trade-offs may be desirable at different stages of the design. For instance, early in the design process design engineers may value faster turn-around time at the cost of some quality, but at later stages (e.g. during final timing closure) may be willing to spend additional run-time for improved quality. Therefore the entire run-time/quality trade-off curve of a placement algorithm is of interest. Figure 6 shows the run-time versus quality trade-offs for WL and CPD achieved by the different algorithms.

For the VTR 8 placer, this was achieved by varying the number of moves per temperature (M in Algorithm 1), which is the best run-time/quality trade-off parameter as determined by the original developers. As M is decreased the placer performs fewer moves which decreases run-time, at the cost of some QoR. While the quality degradation is initially moderate it rises quickly for run-time below 0.4, as the solution space is not being explored sufficiently to find a good solution.

Our placer using the RL-enhanced move generator (RL Agent) outperforms the VTR 8 placer, achieving a better trade-off – particularly at low run-times. This was achieved by varying ε and γ at the same values of M used by the VTR 8 placer. The RL Agent curve is shifted left (better run-time) and below (better quality) the VTR 8 curve, running up to $2\times$ faster at equivalent quality.

The improvements are more pronounced at low run-times where a small number of moves are being made. Here the RL Agent significantly out performs VTR 8, achieving results which are otherwise unachievable by the VTR 8 placer in the same run-time budget. The RL Agent is able to focus the placer's efforts on high reward moves, which is particularly valuable when only limited exploration is possible. At higher run-times, the improvement is less pronounced, likely because enough moves are performed for the standard placer to sample these high reward moves.

To verify our agent is responsible for these improvements Figure 6 also includes results for a 'Random Agent' which uses the same move generator, but randomly selects the type of move to perform (i.e. ignores any learned action values). The Random Agent performs uniformly worse than both our RL Agent and VTR 8. This shows these improvements are *not* simply due to an improved move generation mechanism, but are a result of the RL agent's ability to learn and adapt as optimization proceeds.

To further illustrate this, we can look at how the agent's move proposal distribution changes as shown in Figure 7. First,

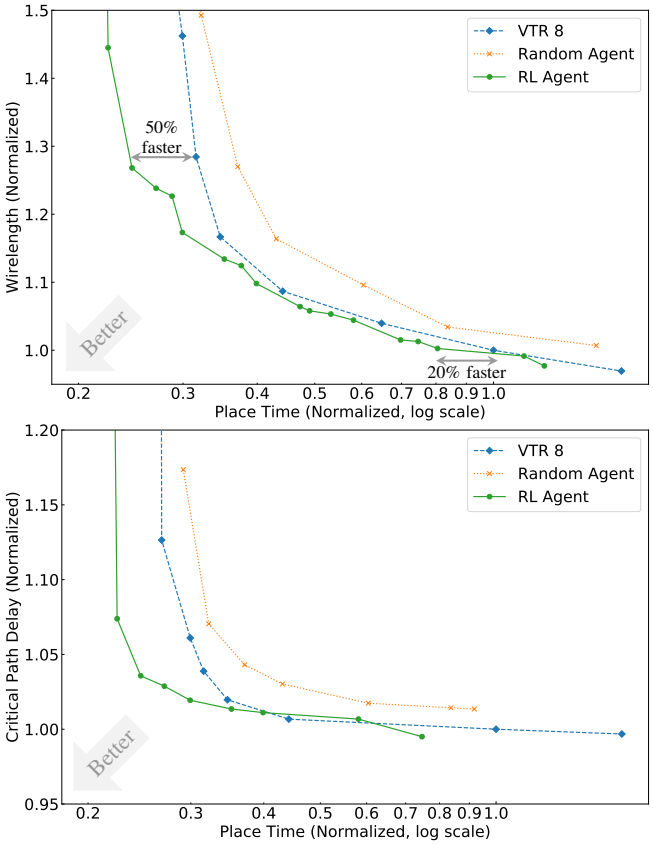

Fig. 6: Pareto optimal run-time versus quality trade-offs on the VTR Benchmarks ($> 10K$ primitives, geomean over 3 seeds). Run-time and quality metrics are normalized to the VTR 8 default settings.

comparing across the different benchmarks circuits we can see the agent chooses to emphasize moves of different types on different circuits. For instance on `mcml` it focuses primarily on moving Logic blocks ($> 80\%$ of all proposed moves), with less than 4% of moves spent on DSP blocks. In contrast, `stereovision2` has a more even distribution, with 33% DSP moves, 48% Logic moves, and 19% IO moves.

We can also look at how the move proposal distribution changes during the anneal, showing how the agent adapts online.[7] For all benchmarks in Figure 7 the placer initially focuses its efforts on IO moves. With `blob_merge` the agent then shifts its focus to Logic moves while still performing a moderate fraction of IO moves. For `LU8PEEng` the agent focuses nearly exclusively on Logic moves during the middle portion of the anneal before it starts performing more DSP and IO moves towards the end of the anneal. `stereovision2` shows somewhat different behaviour, with the agent splitting its efforts between Logic and DSP through out the anneal with an increased focus on DSPs near the end.

VI. Conclusion & Future Work

In conclusion, we've presented a RL-enhanced FPGA placer which uses a RL agent to control how the placer explores the solution space. The agent controls a more flexible SA move generator, and learns online what types of moves are productive.

[7]In contrast, the move proposal distribution of the standard VTR 8 placer is fixed and statically determined by the frequency of block types in the netlist.

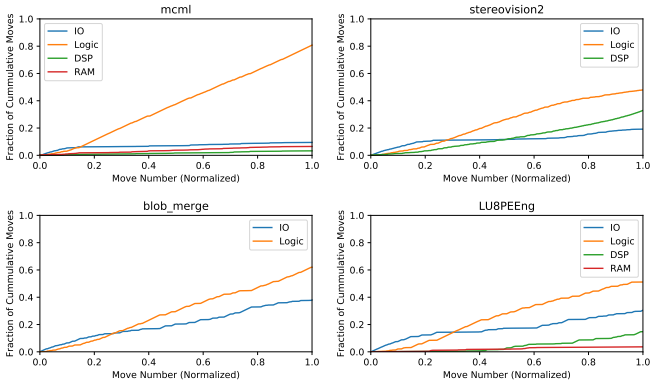

Fig. 7: Cumulative number of moves performed during the anneal for several benchmark cicruits. The RL-enhanced move generator adapts online to both benchmark characteristics and optimization progress.

This enables the placer to adapt based on circuit characteristics and how optimization progresses.

Compared to the VTR 8 placer, our result show the RL-enhanced placer achieves an improved run-time/quality trade-off. This allows our placer to achieve the same QoR while running up to $2\times$ times faster. Our approach is particularly effective in a low run-time regime where few moves are made, and performing 'good' moves is more beneficial.

This illustrates how RL can be used to make more adaptable CAD tools and learn effective optimization heuristics with reduced human design effort. RL offers a very general framework for formulating these types of problems and so should be applicable at various stages of the CAD flow.

There are many potential avenues for future work. While we showed our technique was effective with a relatively small number of simple moves (based on block type) we believe using a wider variety of more powerful moves will make the techniques presented here more effective, as they will give the agent a broad range of capabilities which it can use to improve placement quality.

The reward formulations (Section IV-A) have also not been well explored and can likely be improved. One factor not accounted for is the run-time impact of different move types, as some moves may be more computationally intensive than others. We would like the agent to account for this, enabling it to effectively trade-off both quality and run-time.

Currently our agent has only a single state (Section IV-B), which limits its ability to learn longer time scale phenomena and situationally dependent behaviour. Additional state information (e.g. circuit and optimizer statistics) would likely help it make better decisions. The agent also uses ε-greedy action selection (Section IV-C). One drawback of this approach is if several actions are perceived to have nearly equal value, the agent will usually greedily select the highest value action. A less greedy approach to action selection (e.g. soft-max) would likely better balance the agent's actions with its perceived action values.

Our agent also only learns online. While we believe online learning is important (since it allows the agent to adapt to the current problem characteristics), it means the agent begins learning from scratch at the start of each placement. We expect off-line training (e.g. on a suite of benchmark circuits) would help the agent learn general techniques based on a wider range of experience which would further improve results. Online learning could then be used to ensure the agent still adapts to the specific placement problem.

Additionally, our agent uses relatively simple K-armed bandit-based RL algorithms. While these approaches are fast and run-time efficient (important since they are used in the placer's inner loops) there are a wide variety of more powerful RL algorithms which could be applied, such as Temporal Difference Learning and Policy Gradients [8]. We expect more complex RL techniques will need to be invoked periodically (to keep their run-time overheads low) with the goal of setting the general optimization direction. This direction can then be fine-tuned online with fast RL algorithms like those used in this work.

Finally, we believe RL techniques can be used with other CAD algorithms (e.g. analytic placement, negotiated congestion routing), however the application of RL to such algorithms remains future work.

Acknowledgments

The NSERC/Intel Industrial Research Chair in Programmable Silicon, Huawei, the Canadian Foundation for Innovation, and an Ontario Graduate Scholarship supported this work.

References

[1] C. Xu, G. Liu *et al.*, "A Parallel Bandit-Based Approach for Autotuning FPGA Compilation," in *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, 2017, pp. 157–166.

[2] N. Kapre, H. Ng *et al.*, "Intime: A machine learning approach for efficient selection of fpga cad tool parameters," in *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, 2015, pp. 23–26.

[3] J. Kober and J. Peters, *Reinforcement Learning in Robotics: A Survey*. Springer International Publishing, 2014, pp. 9–67.

[4] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *ArXiv*, vol. abs/1611.01578, 2016.

[5] L. Li, W. Chu *et al.*, "A contextual-bandit approach to personalized news article recommendation," *Int. Conf. on World Wide Web*, 2010.

[6] D. Silver, T. Hubert *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[7] O. Vinyals, I. Babuschkin *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, 2019.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

[9] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, 1997, pp. 213–222.

[10] M. Hutton, V. Betz, and J. Anderson, "FPGA Synthesis and Physical Design," in *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*, 2nd ed. CRC Press, 2016, ch. 16.

[11] K. E. Murray, O. Petelin *et al.*, "VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling," *ACM Transactions on Reconfigurable Technology Systems*, 2020, To Appear.

[12] K. Vorwerk, A. Kennings, and J. W. Greene, "Improving simulated annealing-based fpga placement with directed moves," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 2, pp. 179–192, 2009.

[13] A. Ludwin and V. Betz, "Efficient and Deterministic Parallel Placement for FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, pp. 22:1–22:23, 2011.

[14] I. L. Markov, J. Hu, and M. Kim, "Progress and Challenges in VLSI Placement Research," *Proc. of the IEEE*, vol. 103, no. 11, pp. 1985–2003, 2015.

[15] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, "Liquid: High quality scalable placement for large heterogeneous FPGAs," in *Int. Conf. on Field Programmable Technology*, 2017, pp. 17–24.

[16] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous FPGAs," in *Int. Conf. on Field Programmable Logic and Applications*, Aug 2012, pp. 143–150.