Learn to Place: FPGA Placement Using Reinforcement Learning and Directed Moves

Mohamed A. Elgammal[1,2], Kevin E. Murray[1] and Vaughn Betz[1,2,3]
[1] *Dept. of Electrical & Computer Engineering, University of Toronto, Canada*
[2] *International Centre for Spatial Computational Learning (SpatialML)*
[3] *Vector Institute for Artificial Intelligence, Toronto, Canada*
mohamed.elgammal@mail.utoronto.ca
{kmurray,vaughn}@ece.utoronto.ca

***Abstract*—Simulated Annealing (SA) is widely used in FPGA placement either as a standalone algorithm or a refinement step after initial analytical placement. SA-based placers have been shown to achieve high-quality results at the cost of long runtimes. In this paper, we propose an improvement of SA-based placement using directed moves and Reinforcement Learning (RL). The proposed directed moves explore the solution space more efficiently than traditional random moves, and target both wirelength and timing optimizations. The RL agent further improves efficiency by dynamically selecting the most effective move types as optimization progresses. Taken together, these enhancements allow more efficient exploration of the large solution space than traditional annealing. Experimental results on the VTR benchmark suite show that our technique outperforms the widely-used VTR 8 placer across a wide range of CPU-quality trade-off points, achieving 5-11% reduced wirelength and comparable or shorter critical path delays in a given runtime, or 33-50% shorter runtimes for a target quality point.**

***Index Terms*—FPGA, placement, directed moves, reinforcement learning**

I. Introduction

FPGA capacity continues to scale with Moore's law, with Xilinx recently announcing an FPGA that contains 9 million logic cells and almost 35 billion transistors [1]. While ever-larger FPGAs have allowed much more capable systems based upon them, they also put pressure on the computer aided design (CAD) flows used to implement ever-larger designs. The reprogrammability of FPGAs allows fast realization and in-system debugging of a hardware design, greatly improving designer productivity, but this advantage is undermined if CAD tool runtime is long or the result quality is poor. Consequently, a major goal for the FPGA industry is to devise new CAD algorithms that can produce high-quality results for large designs at reasonable runtime.

Placement is a crucial stage in FPGA CAD flow. It is typically the most time consuming portion of the FPGA compilation flow [2], with a runtime of several hours or more for large designs. Moreover, it greatly affects the QoR of the entire flow. If the placement algorithm produces a solution that requires too much wiring the router will fail – or take a long time – to find a solution, as routing resources in an FPGA are pre-fabricated and hence of limited capacity. Also, placement greatly impacts the final design speed as the best-case routing delays are determined by placement. Final timing optimizations like buffer insertion and gate resizing are not possible in an FPGA, so a placement that forces slow wiring on the critical path cannot meet timing regardless of the effort expended later in the flow. While FPGA placement shares some similarities with ASIC placement, it is a more constrained problem as both the locations where blocks of different types can be placed, and the wiring patterns allowing interconnection of these blocks are pre-fabricated.

Historically, three main techniques have been used to attack the placement problem: partition-based, analytical placement (AP), and iterative improvement (especially simulated annealing) techniques. Partition-based techniques [3] can produce a solution in a short runtime, but they suffer from low QoR. Analytical Placement (AP) approximates the placement goal as a smooth function and finds block locations that minimize this function under a set of constraints that are gradually refined to guide the solution to a mostly legal placement. Analytic techniques are used in several FPGA academic placers, such as Gplace [4] and UTPlaceF [5]. However, to fully optimize an FPGA design, analytical placers generally need a detailed improvement (i.e. refinement) step. These detailed placers are often implemented either as a low temperature simulated annealing algorithm, or as greedy block moves (as in HeAP [6] and RippleFPGA [7]) that can be considered a zero temperature (quench) simulated annealing algorithm. The Liquid [8] placer combines analytic placement with simulated annealing to legalize the locations of "hard" blocks (e.g., RAMs, DSPs). With the exception of Liquid, which combines aspects of simulated annealing for hard blocks with an analytic engine, academic analytic placers for FPGAs have focused on routability and are not timing driven.

Simulated annealing is also used as a standalone placement algorithm for FPGAs, as it naturally handles the constraints on legal placement and the complex non-smooth routing delay vs. net terminal location functions in FPGA architectures. For example, VPR [9] is a widely used academic place and route tool which uses simulated annealing based placement to adapt to a variety of FPGA architectures, and hence is used to explore and evaluate FPGA architecture enhancements. VPR is also used as the production CAD flow to program both novel research FPGAs and some commercial FPGAs with an open-source flow [10]. The commercial Intel Quartus placer also leverages simulated annealing [11]. Simulated annealing plac-

ers are known to achieve high-quality results at the expense of long runtime.

As simulated annealing is the backbone of many placement approaches and the detailed improvement step of other ones, our work increases its efficiency by proposing new directed moves that can efficiently explore the solution space in reasonable runtime. However, scheduling the use of different directed moves for different FPGA architectures, different circuits and also different stages of the placement task is a challenging task. Consequently, we want our algorithm to "learn" the most effective perturbation using reinforcement learning (RL).

In this paper, we propose multiple directed moves that target optimizing both wirelength and critical path delay. Then, we leverage RL techniques to dynamically select the most effective move types throughout different placement stages, on different circuits and FPGA architectures. Taken together, these techniques help the annealer rapidly and efficiently explore the large placement solution space, allowing simulated annealing to obtain high-quality results in reasonable runtime.

II. Related Work

Prior research focused mainly on optimizing the runtime of simulated annealing (SA) in FPGA placement through two productive approaches: 1) Parallel move generation and 2) Directed search of the solution space.

One way to reduce SA runtime is to leverage multiple CPU cores. Ludwin et al. [12] performed multiple moves (placement perturbations) in parallel in the commercial Quartus placement algorithm, while preserving both algorithm determinism and achieving quality of results equivalent to a serial annealer. An et al. [13] explored related techniques in the academic VPR annealer, and showed that speedups of $5.9\times$ were achievable while maintaining quality and determinism. Goeders showed that further parallelism could be exposed if the proposed moves were constrained to optimize different regions of the chip, obtaining higher speedups, but at a cost of 10% quality loss. [13], [14]. The techniques we explore in this paper are complementary to these parallel techniques, as the more effective moves we create and select could also be performed in parallel.

A second technique to speed up SA is to use directed moves. Vorwerk et al. [15] proposed multiple strategies to create moves that more efficiently search the FPGA placement solution space for better timing and wirelength results. They calculated the effectiveness of each move and used this effectiveness to determine the probability of doing it in the future. Altera's Quartus placer also uses more than 10 different types of directed moves but to our knowledge, no details about the exact directed move types employed or how they are selected has been published [11]. RBSA [16] is another SA-based placement engine that proposes "range-based" moves that limit the swap operations to a range of locations around the center of a net that connects to the moving block. It also prioritizes badly placed nets to be chosen to move more frequently. It was further extended in ARBSA [17] to enable timing-driven optimizations. Our work is in the same broad space as these prior studies, but we explore many different move types. To guide the annealer in choosing these types, we explore reinforcement learning approaches that have not been previously considered in this domain.

On the other hand, RL techniques have received significant attention recently due to their superior performance in many applications like website recommendation [18] and decision-making games (e.g. Go and chess [19]). The use of RL in electronic design automation has also been previously explored.

Mirhoseini et al. [20] trained an RL agent to floorplan an ASIC netlist onto a chip canvas. They trained a deep neural network to be able to predict the reward across a wide variety of netlists and global placements. Then, the RL agent uses this predicted reward value to choose an optimized floorplan. After offline training on several problems, this technique achieved superior performance on a specific netlist after 6 hours of further refinement.

In FPGA placement, Murray et al. [21] proposed using a simple single-state RL agent to choose which type of block to move during the anneal, but still always moved the block randomly. They showed that their pareto-optimal results outperformed VPR 8 in terms of QoR achieved in a given runtime.

III. Background

A. Reinforcement Learning

Reinforcement Learning (RL) is a Machine Learning (ML) approach that has led to high-quality results in a wide variety of domains [22]. RL techniques depend on having a software agent that, given the current environment state, chooses to take one of multiple actions to maximize the total reward. The agent works in a closed loop as shown in Fig. 1. It interacts with an environment by performing an action a from a set of the available actions A. Then, the agent interprets how the environment reacts to the chosen action and determines the reward of this action r_{t+1} and the state of the next timestamp s_{t+1} from a set of the available states S. Any action chosen will not only affect the immediate reward but also all the upcoming rewards and states. RL techniques rely mainly on action values. The action value $Q(s, a)$ is the immediate and future expected reward if the action (a) is chosen in a state (s). The two main considerations for any RL algorithm are:

- Action value estimation: How Q is estimated from the agent's experience
- How Q is used to select the next action

A well-known special case of the RL problem is called the K-armed bandit problem where there is only a single state. Hence, Q becomes a function of the actions only: $Q(a)$.

One method to estimate Q values is the tabular method in which a tabular Q function 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:

$$Q(a_{t+1}) = Q(a_t) + \alpha \times (r_{t+1} - Q(a_t)) \quad (1)$$

where α is the step size, and the term $(r_{t+1} - Q(a_t))$ represents the error in agent's estimation of the action value.

When the number of actions and/or states increase, storing Q in a table becomes infeasible and is usually approximated,

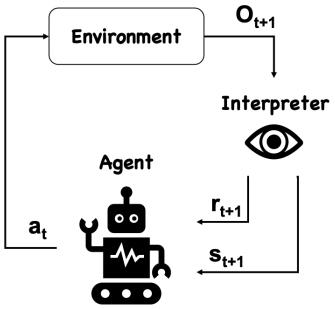


Fig. 1: Reinforcement learning strategy.

using a deep neural network in Deep Reinforcement Learning (DRL). In that case, the agent's experience is used to tune the neural network weights.

B. Simulated Annealing Placement

SA [23] is one of the most commonly used heuristics to solve the placement problem. SA starts with an initial solution and repeatedly generates perturbations (or *moves*) based on the current solution. It estimates the change in the solution cost to decide if the move is accepted or not. As SA is based on the idea of annealing in metallurgy, it starts at a high temperature where all the blocks have high energy (i.e. high probability of accepting any move). Gradually, the temperature drops, and so does the probability of accepting a move –that increases the placement cost– decreases. When the temperature becomes zero, only greedy moves that decrease cost are accepted. This technique can escape local minima as it accepts some moves that increase the cost (i.e. hill climbing). However, it needs a large number of moves to explore large solution spaces. Algorithm 1 shows the SA procedure. The QoR and runtime of SA is strongly affected by the annealing schedule, which determines how the temperature is decreased and the probability of accepting moves.

In FPGA placement, a perturbation (move) means changing the location of some netlist blocks. One of the simplest move types is a *swap*, which swaps two blocks of the same type. As shown in Fig. 2, FPGAs are heterogeneous, with different locations in the FPGA resource grid being able to accommodate different block types, such as IOs, CLBs, DSPs and BRAMs. VPR [9] uses simple swap-based moves. However, there are more powerful moves that exploit information about the netlist and the structure of the problem to move blocks towards their optimal locations.

IV. Directed Moves

As we mentioned previously, the main drawback of the SA placement algorithm is its long runtime to converge to a solution with acceptable QoR, particularly as design size and hence solution space is rapidly increasing for FPGAs. VPR8 [9] uses SA with a random swap-based move generator. It chooses a block (or group of blocks that must move as a group, such as a carry chain) randomly from the netlist. Then, it chooses a random location of the same type within a range

Algorithm 1: Simulated annealing (SA) algorithm

Input: Initial Placement P_{init}, Initial temp T_{init}, M moves per temp.
Result: An optimized placement
$P \longleftarrow P_{init}, T \longleftarrow T_{init}$
while *!(ExitCondition*(P, T)*)* **do**
  for $1...M$ **do**
    $P' \longleftarrow GenerateMove(P)$
    $\Delta cost \longleftarrow EvaluateMove(P, P')$
    if *isAcceptedMove*$(\Delta cost, T)$ **then**
      $P \longleftarrow P'$
    end
  end
  $T \longleftarrow Update\_Temp(T)$
end
return P

around the original location in the device. If the location is empty, it moves the block to this location. If another block was already placed in the proposed location, the two blocks are swapped. Finally, it estimates the change in timing (based on best-case routing delays and the timing criticality of the affected connections) and wirelength (based on the change in net bounding boxes) costs to assess the move and determine whether to accept or reject it.

To improve convergence and quality, we propose several directed moves that help to explore the solution space more efficiently. Ideally, these moves should not only propose useful changes, but also be fast to perform as they will run thousands to millions of times. In the following subsections, we will discuss the moves we used in our proposed engine.

A. Median Move (Med.)

This move was initially proposed by Vorwerk et al. [15]. It mainly targets wirelength minimization by moving a block (b_i) into the *median region* of this block which is the range of locations (x, y) of positions that minimize the HPWL. The median region is calculated using Algorithm 2. The main differences between this algorithm and the one proposed in [15] is that in the *get_bounding_box()* function, we are using an incremental bounding box calculation technique similar to [24]. Moreover,

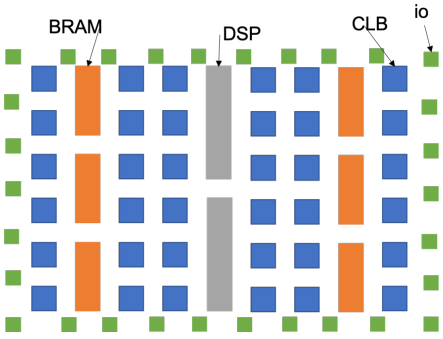


Fig. 2: FPGA location grid.

Algorithm 2: Median move algorithm

Choose a random block b_i
Define empty vectors X_{coords}, Y_{coords}
for P_i *in block_pins*(b_i) **do**
  n_i is the net connected to pin P_i
  if n_i *is not a high fanout net* **then**
    x_l, x_r, y_l, y_h = get_bounding_box(n_i)
    push x_l, x_r in X_{coords}
    push y_l, y_h in Y_{coords}
  end
end
Sort X_{coords}, Y_{coords}
x_1, x_2 = the two middle elements in X_{coords}
y_1, y_2 = the two middle elements in Y_{coords}
Define the median region to be $x_1 < x < x_2$, $y_1 < y < y_2$
Choose a random location within the median region with the same type as b_i

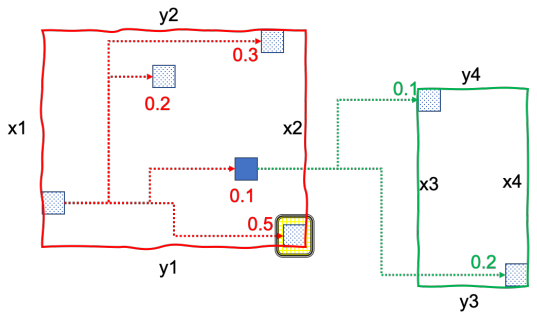


Fig. 3: Edge-Weighted Median move example. The solid box is the moving block, the dotted boxes are the fanin and fanout blocks, the numbers are the timing criticalities of various connections, dotted lines are the connections and the solid line represents the bounding box of each net. The final Edge-Weighted location is the yellow colored region inside the black box.

we exclude nets with fanout higher than "***high_fanout_limit***" terminals to speed up the calculation. However, we found that in many cases the median region is too small. Therefore, to avoid becoming stuck and proposing the same move repeatedly, we allow the block to move to a location in a range around the center of median region. We empirically found that allowing an initial range of "***max_range_limit***" units in any direction around the median range that gradually shrinks to 1 unit late in the anneal is a good choice. This range is applied to the compressed device grid of the block type [9] which enables the sparse blocks to move in a different columns even late in the anneal.

B. Edge-Weighted Median Move (E.W.Med.)

Vorwerk et al [15] proposed constructing a median region as described in the prior section, but then weighting (shifting) the coordinates by net criticality in an attempt to improve circuit timing. The criticality of any connection is a value $\in$ [0,1] where connections with criticality 1 have 0 slack, and those with criticality 0 have a large positive slack. We found this formulation did not achieve good results. Instead we developed a new formulation that constructs target regions from criticality-weighted *edges* of the net bounding boxes, which we call edge-weighted median and describe below.

Fig. 3 shows an edge-weighted median move. We are moving one block (the blue square) connected to two nets (the green and red dotted lines). The sinks of each connection have their timing criticality values labelled. The bounding box of each net is outlined by the solid colored box with the same color as its net. In the median move, x_1, x_2, x_3, x_4 are all weighted equally and the same for y_1, y_2, y_3, y_4. In the edge-weighted median move, however, each coordinate is weighted by a different factor. In this example, x_1 will be weighted by one, which is 10 times the criticality of the connection from the red net driver to the moving block (0.1). x_2 will be weighted by 5 as this edge of the red net bounding box is determined by a pin on a connection with timing criticality of 0.5. Following the same strategy, x_3 and x_4 will be weighted by one and two correspondingly, and the Y_{coords} are computed similarly. We then calculate the median of the weighted values to extract the edge-weighted median region, which in this case is a single location: $(x2, y1)$. Edge-weighted median can simultaneously optimize timing and wirelength by shrinking net bounding boxes with a preference to move closer to timing-critical pins that are setting the net bounding box limits. As with the median move, we choose a random location from a range around the center of the edge-weighted region.

C. Centroid Move (Cen.)

The centroid move is inspired by analytic placers which model net connections as springs and calculate the point of force equilibrium. It is also similar to the force-directed (FD) moves used in Hentschke's greedy perturbation placer [25] for ASICs. For a chosen block, this move calculates the forces/weighs acting on the block based on its connections to other blocks. We weight each connection to the moving block by 1. We then calculate the zero-force position using equation (2) where $x_{d(i)}$ is the x location of the driver of net i, $y_{d(i)}$ is the y location of the driver of net i, $sinks(i)$ is a list of the sink pins of net i, W_j is the weight of the connection between the moving blocks and the pin j. Y_c is calculated similarly.

We found that moving only to the single location calculated in this way is too restrictive: there may be no legal location for the block type at that spot, or we may propose too many moves to the same spot and not explore the space adequately. Consequently, we define a small region around the zero-force location (as described earlier for the previous moves) and choose a random compatible location within it.

$$X_c = \frac{\sum_{i \in \text{in nets}} x_{d(i)} * W_{d(i)} + \sum_{i \in \text{out nets}} \left(\sum_{j \in \text{sinks}(i)} x_j * W_j \right)}{\sum_{i \in \text{in nets}} W_{d(i)} + \sum_{i \in \text{out nets}} \left(\sum_{j \in \text{sinks}(i)} W_j \right)} \quad (2)$$

D. Weighted Centroid Move (W.Cen.)

Weighted centroid move is similar to the centroid move, but weighs each connection by its criticality, $W_i = Crit_i$.

E. Critical Random Move (C.R.)

Another move that can improve the critical path delay of the placed netlist is the critical random move. This move chooses blocks with highly critical connections and moves it to a compatible random location in a range around its original location; it was combined with frequent timing analysis in [26] to improve FPGA timing.

This move relocates blocks that have at least one connection with timing criticality $>$ "***criticality_limit***". An advantage of this move is that it requires minimal extra computation which means it is fast to propose and evaluate. At the same time, it focuses more on moving highly critical blocks, increasing the probability of finding a good placement for the speed-limiting parts of a circuit.

F. Feasible Region Move (F.R.)

Chen et al. [27] defined a Feasible Region (FR) where a block can be placed to minimize the critical path delay. Vorwerk et al. used the same idea in their MPD move [15]. Similarly, we propose a feasible region move which moves a highly critical block to a random location inside its feasible region. First, we choose a random block with highly critical connections to move. Then, we identify the highly critical inputs to the block and the most critical output. Finally, we calculate the feasible region using the algorithm proposed in [27]. While this move takes more computation than the critical random move, it directly minimizes the delay of the critical connections.

G. Random Move (Rand.)

We have also preserved the original VPR 8 move which we called the "Random Move (Rand.)" in our analysis. Early in the anneal, this move will move a randomly chosen block to any legal location in the chip with equal probability. Based on the acceptance rate of moves, the range (x,y distance) a block can move to gradually shrinks until it is only able to move to locations one unit away (for that type of block) in each dimension.

V. Reinforcement-Learning-Based Placement

After defining and implementing the directed moves in Section IV, we next determine how to use them effectively. In particular, we need to find answers to the following questions: When should each move type be used? What should be the probability of each move occurring? Should these probabilities be fixed for all designs and FPGA architectures? Should they even be fixed throughout the annealing process?

Answering these questions is challenging for the group of moves proposed above and becomes even more challenging as the number of move types further increases (we believe that many more types of perturbations can be proposed). We believe the answer will be highly situational. For instance, the most effective moves will depend on the netlist we want to place, the current placement progress and the target architecture. Due to the large number of factors that affect the most effective moves and with a multiplicity of available moves, it would be challenging or impossible for a human expert to create a fully tuned schedule of moves to propose at different points in time.

As a result, we leverage a reinforcement learning agent to *automatically* adapt the probabilities of using each move according to each situation. In the next subsection, we will discuss the formulation of our problem into an RL problem, how we addressed the exploration vs. exploitation issue and how we designed the reward function.

A. Problem Definition

As we discussed in Section III, there are two techniques used in RL to estimate state action value $Q(s, a)$: tabular methods and function approximation methods (DRL). The choice between these approaches depends mainly on the problem state and action spaces.

Murray et al. [21] used a tabular method with a single state and an action space of moving different block types randomly. We expand the action space to the larger variety of move types detailed in Section IV. We also introduce multiple states that represent the stage of the annealer. As shown in Table I, there are seven actions corresponding to the seven moves discussed in Section IV. The state space comprises two states which represent the stage of the annealing process. S_0 represents the earlier part of the anneal, when large changes are being made and the focus is on obtaining a good global placement that optimizes wirelength and has reasonable overall timing. Late in the anneal there is more visibility into the true critical paths of the design, and additional focus on timing optimization is appropriate. Consequently, in the first state only the first 4 actions are available as they primarily target wiring and coarse timing optimization. We determine that we have moved into the second (S_1) state by monitoring the rate of temperature update in VPR's adaptive annealing schedule; when the new temperature is at least 30% smaller than the prior temperature, placement is no longer performing much hill climbing, indicating we are in S_1. In the S_1 state, the RL agent will select amongst all 7 move types, including the final 3 in Table I (edge-weighted median, feasible region, and critical random) which focus more on timing optimization.

Having only two states makes it feasible to use tabular methods to estimate state-action values, and is similar to having two K-armed bandit agents (one per state). In detail:

- The agent chooses an action $a_t \in A$ (a move type).
- The move is evaluated, blocks are swapped and the change of the placement cost is calculated as Δbb_{cost}

The actual research question.

TABLE I: RL agent actions and state space

Space	Values
Actions (A)	{R, Med., Cen., W.Cen., E.W.Med., C.R., F.R.}
State (S)	{S_0, S_1}

(the normalized change of wirelength cost), Δt_{cost} (the normalized change of timing cost) and $\Delta cost$ (the total change of cost) [28].

- The annealer decides whether to accept or reject the move.
- The agent recieves a reward r_{t+1} to update the Q values based on the calculated $\Delta costs$, thereby influencing future action selection.

In our case, the problem is affected by both static characteristics like circuit structure and FPGA architecture, and dynamic characteristics like the types of moves that are most effective and the current placement quality which are evolving throughout the optimization process. Hence, our problem is non-stationary and the agent should pay more attention to more recent rewards as they better reflect recent changes in the environment. Consequently, as in [21], we set α (in Equation 1) according to the exponentially weighted formula as follows:

$$\alpha = 1 - e^{log(\gamma)/M} \tag{3}$$

where M is the number of moves per temperature, and γ is a hyper-parameter that controls the fraction of weight given to moves which occurred $> M$ moves ago (effectively controlling the length of the agent's memory).

B. Exploration vs Exploitation

Once the agent updates the Q values, it selects the next action to perform. The agent faces two contradictory choices:

- exploit the optimal action from its knowledge (i.e. the greedy action). This selection will maximize the immediate reward, or
- explore other non-greedy actions to improve its estimation of the Q values.

This issue is particularly challenging in non-stationary problems, where what the agent may have previously learned as the best choice may become sub-optimal later. There are many techniques to overcome this challenge [22] such as ε-greedy and softmax formulations. As it is likely that multiple types of moves are productive at a certain placement state, we prefer softmax, which allows exploitation of multiple move types that are performing well.

In softmax, the agent chooses moves propotional to their state action values (Q). Hence, the greedy-action would be more likely to be chosen, and other near-greedy actions will have comparable probabilities. Applying the common softmax formula to calculate the probability for each action yields:

$$P_a = \frac{e^{Q(a)}}{\sum_{i=1}^{k} e^{Q(i)}} \tag{4}$$

However, as the reward values are too small (in the range of $1e{-}9$), we used the Boltzman softmax implementation instead to sharpen the exponential function as shown in Eq. 5, where β is a parameter that controls the sharpness of the exponential function [29].

$$P_a = \frac{e^{\beta * Q(a)}}{\sum_{i=1}^{k} e^{\beta * Q(i)}} \tag{5}$$

To force exploration of all actions, we include a minimum probability for all actions (α), which enables exploration and is particularly important early in the anneal:

$$P_a = \frac{max(e^{\beta * Q(a)}, \alpha)}{\sum_{i=1}^{k} max(e^{\beta * Q(i)}, \alpha)} \tag{6}$$

C. Reward Function

To help the agent interpret the environment's output, we need to define a *reward function* which translates the output O_{t+1} into a reward r_{t+1}. The agent then tries to maximize its cumulative reward. Crafting a good reward function is important, since it is the only feedback the agent receives to adjust its behaviour.

To capture our intent (a high quality optimization result in reasonable runtime), we consider the change of the placement cost ($\Delta cost$) which includes both timing and wirelength components. As the placement problem is a minimization problem but the agent's target is to maximize its reward, we use the negative of the cost change in any reward function we propose.

The first reward function we consider [21] is:

$$r_t = -\Delta cost \tag{7}$$

This reward function seems to match our intent of minimizing cost. However, it penalizes the agent with a large negative reward for moves that increase the cost (hill climbing moves). Hence, we also consider the following reward function:

$$r_t = \begin{cases} -\Delta cost, & \text{if } \Delta cost < 0 \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

which was shown to be effective in [21].

However, while the above reward functions capture the goal of generating high-quality placements, they do not capture any metric of the runtime to a solution. To make the agent runtime aware, we propose a reward function of:

$$r_t = \begin{cases} \frac{-\Delta cost}{t(i)}, & \text{if } \Delta cost < 0 \text{ for move type } i \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

where $t(i)$ is a static move-dependent normalizing factor which represents the average time consumed to evaluate moves of type i. We pre-calculate $t(i)$ by averaging the runtime of each move type over many circuits. With this reward function, the agent prefers the move type that maximizes the rate of quality improvement, thereby minimizing runtime.

We also explored reward functions that give the agent some information about the state of the placement. For example, we hypothesize that moves that enhance wirelength should be more beneficial early in the anneal, while moves that optimize timing are more important late in the anneal. This is due to the fact that timing cost is a maximum of many values while

the wirelength cost is a summation. To make the agent aware of this trait and allow it to choose more wirelength-optimizing moves early in the annealer and more timing-focused moves later, we proposed a reward function that includes the different components of the change of the placement cost (Δbb_{cost}, Δt_{cost}) [28] as shown in Eq. 10

$$r_t = \begin{cases} \frac{-1}{t(i)} \Big((1 + R_s) * \Delta bb_{cost} + \\ \qquad (2 - R_c) * \Delta t_{cost} \Big), & \text{if } \Delta cost < 0 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

where R_s is a *reward shaping* factor that changes throughout the anneal in $[0, 1]$. It starts at 1 and decays as the temperature decreases, guiding the agent to gradually shift focus from wirelength to timing optimization as the anneal progresses.

VI. Results & Discussion

We implemented the proposed algorithms in VPR 8 [9] and compare to the original VPR 8 placer. The VTR benchmark suite [9] was used for tuning and testing the proposed algorithm after excluding small circuits with fewer than 10K primitives. We targeted the flagship VTR architecture `k6_frac_N10_frac_chain_mem_32K_40nm.xml` [9] which is a realistic FPGA architecture with ten 6-input fracturable LUTs and twenty FFs per logic block, along with RAM, DSP and IO blocks.

CPU runtime and QoR are both important metrics when evaluating and comparing placement algorithms. In most placement algorithms, there are some parameters that can be tuned to control the runtime/QoR trade-off. This is useful as the low runtime results are very important in the design process to guarantee fast turn-around time, while the high runtime (higher quality) results are also important in the final design stages. Hence, the entire runtime/QoR trade-off curve of a placement algorithm is of interest. To get a curve that represents this trade-off, we varied the number of moves per temperature parameter (M in Algorithm 1).

To avoid CAD noise, all results shown in this section are averaged over three different seeds and geometrically averaged over different circuits unless otherwise noted.

TABLE II: Best performing parameter values of the agent and the directed moves

Parameter	context	Value
high_fanout_limit	subsection IV-A	10
max_range_limit	subsection IV-A	3
criticality_limit	subsection IV-E	0.7
agent algorithm	Eq. 6	Softmax
β	Eq. 6	100,000,000.
α	Eq. 6	3
γ	Eq. 3	0.05
agent reward function	Eq. 10	R_s-runtime-aware

A. Choosing the agent and its parameters

To choose the appropriate agent and its parameters, many experiments were performed. For example, to address the exploration vs. exploitation dilemma, we varied the parameters of the agent and its different implementations and picked the best performing set of parameters. Table II shows the values we used for each of the agent and move parameters. The Boltzman softmax agent achieves the best result, and the best reward function is the timing-vs.-wirelength-shaped and runtime-aware reward function of Eq. 10 (R_s-runtime-aware).

B. Comparison to VPR 8

Fig. 4 shows runtime versus quality of results (QoR) trade-offs for wirelength (WL) and critical path delay (CPD) achieved by our proposed technique (directed moves + RL agent) and by VPR 8 on the large VTR benchmarks. It is clear that our runtime/quality trade-off curve is below the VPR 8 curve in most trade-off points. The proposed algorithm achieves average wirelength gains of 11% in the low runtime region and 4.5% in the high runtime region compared to VPR 8. The proposed technique also achieves the same or better critical path delay while reducing runtime by over 33% in the high runtime region. To quantify how much gain comes from directed moves vs. the RL agent, we performed an experiment using a random agent that chooses move types randomly with equal probabilities. Our RL agent achieved 3% less WL and more stable CPD at all runtime points and 2% shorter CPD at the high runtime setting compared to the random one, indicating that it is providing a significant part of the gains.

Table III shows the per circuit results of the proposed technique at both low runtime (M=0.125) and high runtime (M=2) budgets, ordered by ascending circuit size. It shows that our gains at low runtime budgets are larger than those at high runtime budgets; when the anneal is allowed to make fewer moves, choosing those moves wisely is crucial and the RL agent is able to select well. It also shows that the proposed technique scales well with design size. For example, comparing the gains achieved for the smallest design (stereovision1.v) and the largest design (mcml.v) at low runtime budget, we can see that the gains are higher for the larger design. This again points out that the RL agent efficiently explores the solution space; a larger design space with a small runtime budget makes efficient exploration crucial.

C. Agent Behavior

To see how the agent behaves in more detail, Fig. 5 shows the cumulative number of each move type performed throughout the annealing process for two different designs (LU32PEEng, stereovision2). It shows how the preferred move type changes through the annealing process and for the different designs. For example, in the stereovision2 design (bottom sub-figure), the agent starts with equal probabilities for all move types. At mid temperatures the median and weighted centroid moves are preferred. Finally, near the end of the anneal moves within a random local region (R.) moves are the most common, and some Edge-Weighted Median

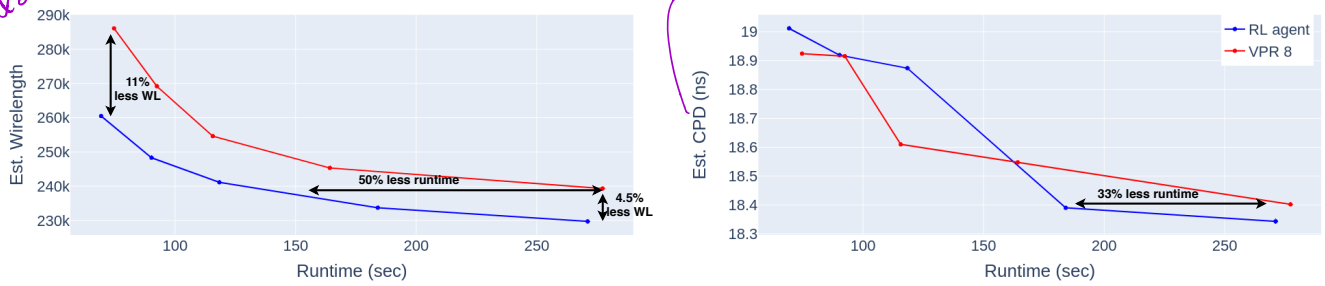


Fig. 4: Comparison of the proposed technique and VPR 8 on large VTR benchmarks

TABLE III: Results of the proposed placement technique on VTR benchmarks designs at low/high runtime budgets (normalized to default VPR 8 between brackets). CPD is critical path delay.

Circuit	Num Primitives	Low Runtime			High Runtime		
		Place Time (s)	CPD (ns)	Wirelength	Place Time (s)	CPD (ns)	Wirelength
stereovision1.v	19549	12.36 (0.91x)	5.68 (1.09x)	86887 (0.99x)	34.8 (1.09x)	5.5 (1.10x)	75165 (0.98x)
stereovision0.v	21789	9.06 (0.89x)	2.87 (1x)	44108.33 (0.93x)	26.2 (1.01x)	2.8 (0.98x)	38964.3 (0.95x)
LU8PEEng.v	33863	60.97 (0.85x)	66.02 (0.98x)	247782.33 (0.91x)	263.5 (0.96x)	65.5 (0.95x)	226943 (1x)
bgm.v	33970	78.43 (0.95x)	19.1 (0.96x)	275143.33 (0.94x)	317 (0.98x)	17.9 (0.96x)	242091.3 (0.96x)
stereovision2.v	42630	149.8 (0.96x)	15.73 (1.06x)	364435 (0.81x)	316.2 (0.93x)	14.6 (0.99x)	306442.3 (0.88x)
LU32PEEng.v	111149	304.62 (0.96x)	65.7 (0.94x)	1155909 (0.98x)	1910.8 (0.82x)	65.2 (0.99x)	1027477.3 (1x)
mcml.v	166678	315.38 (0.96x)	42.23 (1.01x)	739582 (0.83x)	2339.1 (1.08x)	40.5 (1.01x)	666505.3 (0.95x)
Geomean		69.35 (0.93x)	19.01 (1x)	260498.35 (0.91x)	271.1 (0.98x)	18.3 (1x)	229728.5 (0.95x)

moves are also performed. Also, note that the three timing-focused moves (E.W.Med, C.R. and F.R.) are only activated late in the anneal when the agent's state is changed.

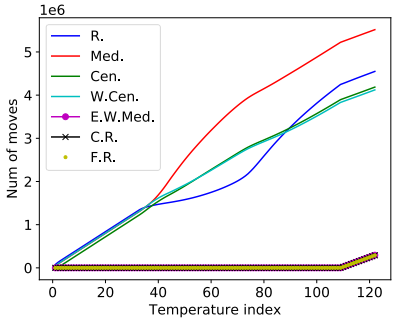
VII. Conclusion

We presented an improvement to simulated annealing placement that leverages multiple directed move types with a reinforcement learning agent controlling which moves to propose throughout the anneal. The proposed algorithm (directed moves + RL) explores the solution space more efficiently than traditional random moves, yielding better QoR at all the CPU runtime points we evaluated. Our technique can achieve 11% less wirelength and the same critical path delay as VPR 8 for low runtime budgets. It also achieves 5% less wirelength and 1% CPD gains at high runtime budgets. In addition, it can achieve comparable QoR to VPR 8 with high effort settings in 33-50% less runtime. The gains of the directed moves with RL algorithm increase as the design size grows, making it a promising technique to deal with the CAD challenge of ever-growing FPGA capacity.

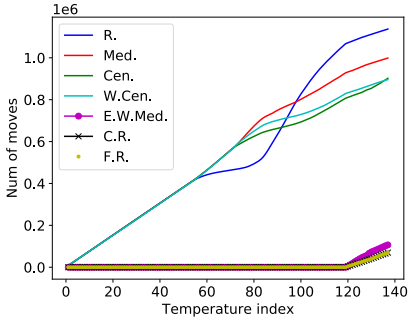
The main contribution of the paper is formulating the FPGA placement problem into an RL problem, and showing this can allow the placement space to be more efficiently searched. There are many directions for future work which build on the RL literature and can enhance this approach. One future direction is to increase the choices available to the RL agent by adding more directed move types. A second direction is to investigate different RL agents, such as agents that use offline learning (pre-trained on a set of benchmarks) to compute initial reward estimates, which are refined by learning on the specific circuit under test during placement.

Acknowledgment

This work was supported by an NSERC/Intel Industrial Research Chair in Programmable Silicon, Huawei and Google.



(a) LU32PEEng



(b) stereovision2

Fig. 5: Cumulative number of each move type proposed vs. annealing progress.

REFERENCES

[1] "Xilinx announces the world's largest FPGA featuring 9 million system logic cells." [Online]. Available: https://www.xilinx.com/news/press/2019/xilinx-announces-the-world-s-largest-fpga-featuring-9-million-system-logic-cells.html

[2] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Titan: Enabling large and complex benchmarks in academic CAD," in *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 2013.

[3] A. Khatkhate, C. Li, A. R. Agnihotri, M. C. Yildiz, S. Ono, C.-K. Koh, and P. H. Madden, "Recursive bisection based mixed block placement," in *Proceedings of the 2004 international symposium on Physical design*, 2004.

[4] T. Martin, D. Maarouf, Z. Abuowaimer, A. Alhyari, G. Grewal, and S. Areibi, "A flat timing-driven placement flow for modern FPGAs," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019.

[5] W. Li, S. Dhar, and D. Z. Pan, "Utplacef: A routability-driven FPGA placer with physical and congestion aware packing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 4, pp. 869–882, 2017.

[6] M. Gort and J. H. Anderson, "Analytical placement for heterogeneous fpgas," in *Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2012.

[7] Chak-Wa Pui, Gengjie Chen, Wing-Kai Chow, Ka-Chun Lam, Jian Kuang, Peishan Tu, Hang Zhang, E. F. Y. Young, and Bei Yu, "Ripplefpga: A routability-driven placement for large-scale heterogeneous fpgas," in *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2016, pp. 1–8.

[8] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, "Liquid: High quality scalable placement for large heterogeneous FPGAs," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017.

[9] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. ElDafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "VTR 8: High Performance CAD and Customizable FPGA Architecture Modelling," *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

[10] "Symbiflow project," 2019. [Online]. Available: https://symbiflow.github.io/

[11] A. Ludwin and V. Betz, "Efficient and deterministic parallel placement for FPGAs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 3, pp. 1–23, 2011.

[12] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008.

[13] M. An, J. G. Steffan, and V. Betz, "Speeding up FPGA placement: Parallel algorithms and methods," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2014.

[14] J. B. Goeders, G. G. Lemieux, and S. J. Wilton, "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition," in *2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2011.

[15] K. Vorwerk, A. Kennings, and J. W. Greene, "Improving simulated annealing-based FPGA placement with directed moves," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 2, pp. 179–192, 2009.

[16] J. Yuan, L. Wang, X. Zhou, Y. Xia, and J. Hu, "Rbsa: Range-based simulated annealing for FPGA placement," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017.

[17] J. Yuan, J. Chen, L. Wang, X. Zhou, Y. Xia, and J. Hu, "ARBSA: Adaptive range-based simulated annealing for FPGA placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 12, pp. 2330–2342, 2018.

[18] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proceedings of the 19th international conference on World wide web*, 2010.

[19] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *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.

[20] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, S. Bae *et al.*, "Chip placement with deep reinforcement learning," *arXiv preprint arXiv:2004.10746*, 2020.

[21] K. E. Murray and V. Betz, "Adaptive FPGA Placement Optimization via Reinforcement Learning," in *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2019.

[22] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[24] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Springer Science & Business Media, 2012, vol. 497.

[25] R. F. Hentschke and R. Reis, "Improving simulated annealing placement by applying random and greedy mixed perturbations [ic layout]," in *16th Symposium on Integrated Circuits and Systems Design, 2003. SBCCI 2003. Proceedings.* IEEE, 2003.

[26] K. E. Murray and V. Betz, "Tatum: Parallel timing analysis for faster design cycles and improved optimization," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018.

[27] G. Chen and J. Cong, "Simultaneous timing-driven placement and duplication," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005.

[28] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for fpgas," in *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, 2000.

[29] P. R. Montague, "Reinforcement learning: an introduction, by sutton, rs and barto, ag," *Trends in cognitive sciences*, vol. 3, no. 9, p. 360, 1999.