Multi-Armed Bandits for Autonomous Timing-driven Design Optimization

Apostolos Stefanidis*, Dimitrios Mangiras*, Chrysostomos Nicopoulos†, Giorgos Dimitrakopoulos*
*Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece
†Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus

***Abstract*—Timing closure is a complex process that involves many iterative optimization steps applied in various phases of the physical design flow. Cell sizing and transistor threshold selection, as well as datapath and clock buffering, are some of the tools available for design optimization. At the moment, design optimization methods are integrated into EDA tools and applied incrementally in various parts of the flow, while the optimal order of their application is yet to be determined. In this work, we rely on reinforcement learning – through the use of the Multi-Armed Bandit model for decision making under uncertainty – to automatically suggest online which optimization heuristic should be applied to the design. The goal is to improve the performance metrics based on the rewards learned from the previous applications of each heuristic. Experimental results show that automating the process of design optimization with machine learning not only results in designs that are close to the best-published results derived from deterministic approaches, but it also allows for the execution of the optimization flow without any human in the loop, and without any need for offline training of the heuristic-orchestration algorithm.**

I. Introduction

Timing-driven design optimization aims at satisfying timing constraints and improving the area and power performance of the design, without affecting its functionality and without violating design rule constraints (e.g., maximum capacitance and maximum slew). Such a multi-objective problem is inherently complex and computationally challenging, since it involves all the steps of the physical synthesis flow [1].

Over the years, multiple optimization methods have been introduced to optimize the design's characteristics and to achieve significant Power-Performance-Area (PPA) improvement. Gate sizing, voltage threshold selection, logic restructuring, timing-driven placement, and buffering insertion/deletion are just a few examples of optimization methods supported by modern EDA tools.

Even if each algorithm is effective in optimizing the design, the *order* of the application of the various algorithms is equally important to the final result. At the moment, the order of application of the available design optimization methods is organized in reference flows based on the experience of the physical design engineers and on how the flow has reacted so far to other designs. If closure is not achieved by the reference flow, the flow is customized for the specific needs of the design. This adaptation is part of a "local" and "online" learning process, whereby the designer observes the deficiencies of the reference flow and customizes it to the specific challenges that he/she faces. Such decision making suffers from the exploitation versus exploration dilemma: exploit the same heuristics that have so far yielded high Quality-of-Results (QoR), or spend some time exploring new heuristics with more uncertain rewards to gather more information and hope to reach a better overall solution in the end.

In this work, we leverage The Multi-Arm Bandit (MAB) model – a primitive form of online reinforcement learning – for the first time (to the best of our knowledge) in an ASIC design flow to autonomously apply design optimization heuristics and achieve timing closure and power/area reductions. In the past, MAB has been successfully used in online recommendations systems [2], [3] and for tuning generic and FPGA compilation flows [4], [5].

The proposed approach tries to answer the following fundamental question regarding design optimization: given a set of well-defined design optimization heuristics with uncertain QoR, and a set of timing constraints, how should one choose the available heuristics online, and without any previous knowledge of the design, to maximize the final QoR over multiple trials while keeping runtime under control? According to MAB, for each one of the applied optimization algorithms, a "reward" is recorded. The MAB policy [6] decides (recommends) which optimization method should be selected next, following a balanced exploitation-exploration approach. The desire to choose a method that has paid off well in the past – thereby offering increased PPA metrics – is balanced with the desire to try different methods that may produce even better results.

The proposed approach has been successfully applied to the ISPD12 and TAU2019 benchmarks in two recent international design optimization contests [7], [8]. In all cases, the MAB-based optimization successfully optimizes the designs and achieves equal, or even better, results than the most efficient methods available in the open literature for the same benchmarks. The most important outcome of this work is that MAB-based design optimization not only achieves timing closure and competitive power/area reductions, but it also does that autonomously without relying on any previous knowledge of the circuits or the optimization methods used, and without requiring any human intervention.

II. Autonomous Online Design Optimization

The proposed methodology automates the application of simpler and more elaborate optimization algorithms to the design under consideration with the goal to autonomously achieve timing closure and improve PPA. The proposed autonomous optimization framework assumes the role of a gambler sitting in front of slot machines (the set of optimization heuristics), who has to decide which machines to play, how many times to play each machine, and in which order, to maximize his/her gain (QoR biased by runtime overhead).

Multi-armed bandits orchestrate the execution of a set of optimization algorithms (as the ones described in Section III), in order to maximize the QoR, driven by the reward function

978-1-7281-2103-1/19/$31.00

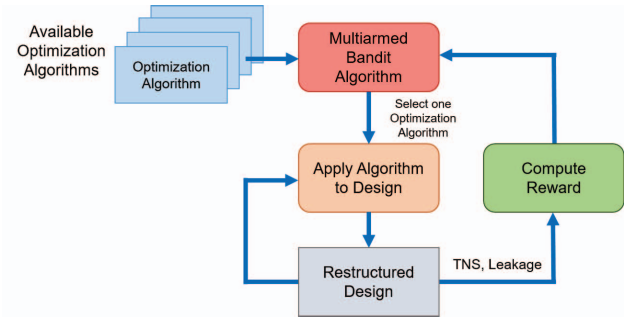

Fig. 1. Autonomous timing-driven design optimization. The design is repeatedly optimized by the algorithm selected by the MAB methodology. The reward of each algorithm and the MAB policy determine which algorithm will be applied next to the restructured and partially optimized design.

defined in Section IV. MAB policy exploits the most effective optimization algorithms already tried, while still exploring new options by activating less frequently used algorithms in the hope of achieving a greater payoff [2], [3].

The proposed optimization flow is shown in Fig. 1. In each round, one optimization algorithm is selected and applied to the design. When the algorithm finishes, its reward is recorded, in order to guide the selection of the following rounds. The changes performed by the algorithm on the design under consideration are kept for the next rounds, unless the obtained result is worse in all aspects (timing, leakage power, and area). As long as the design is altered in each round, successive plays of the same optimization algorithm would yield different rewards. In fact, the rewards are diminishing as timing closure is approached and power/area are already well optimized. The optimization stops when the reward observed in 10 consecutive iterations does not improve by more than 1%.

An algorithm not selected in a specific round is considered "frozen" and neither receives a reward, nor it changes the average reward observed so far.

The MAB methodology takes the average reward observed so far and tries to pick the next optimization heuristic, according to the chosen exploit-vs-explore policy. If you exploit too much, you might miss out on other, really effective heuristics. On the other hand, if you explore too much, you will waste rounds without taking advantage of effective algorithms. Given a history of rewards and a number of "pulls" for each optimization algorithm, the Upper Confidence Bound (UCB) MAB algorithm [6] (the crux of the MAB methodology) calculates an optimistic guess as to how good the expected payoff of each algorithm is, and picks the sequence with the highest estimate as follows:

1) Apply each one of the optimization algorithms once to initialize the mean payoff $\overline{Q}_0(a)$ of each algorithm a.
2) For each next trial t, select the algorithm a to apply in this trial that maximizes $\overline{Q}_{t-1}(a) + \sqrt{\frac{2\log t}{N_a}}$, where t is the total number of trials and N_a is the number of times that algorithm a has been played so far.
3) Observe the reward $Q_t(a)$ of this trial using the reward function described in Section IV, and update the mean reward for this action $\overline{Q}_t(a)$.

The mean reward $\overline{Q}(a)$ for algorithm a can simply be the running average of the rewards gained in the previous applications of the same optimization algorithm $\overline{Q}_t(a) = \overline{Q}_{t-1}(a)\frac{N_a-1}{N_a} + Q_t(a)\frac{1}{N_a}$. Even if this choice seems reasonable, the newly received $Q_t(a)$ reward is scaled by $1/N_a$, which diminishes its effect as N_a gets large. This is problematic, since, as the optimization evolves, the design is gradually moved to a better state and an algorithm tends to be less effective even if, in the first trials, it received high rewards. For example, a method that aims towards improving the Total Negative Slack (TNS) will receive high rewards as the timing improves, but it will receive zero credit once the timing violations are fixed. Therefore, the MAB algorithm will keep picking this algorithm, since its average reward will still be high, and it will require many trials to shift the average reward to a lower value. In order to avoid this effect, we can over-weigh recent rewards over past rewards in order to better highlight the most recent performance of each algorithm [2]. This process is called drifting in MAB-based policies and updates the average reward for each algorithm as follows:

$$\overline{Q}_t(a) = k\,\overline{Q}_{t-1}(a) + (1-k)Q_t(a)$$

The experimental results presented in Section V assume k=0.5. The algorithm stops when a maximum number of iterations is reached (we set it to 200), or when the reward received is less than 0.001 for 10 consecutive iterations.

III. Optimization Algorithms & Heuristics

In this work, we focus on the fine-grained interleaving of buffering and gate-sizing transformations. We define a set of seven design heuristics, with each one focusing on a different aspect of the design. The MAB policy orchestrates the application of the available heuristics with the goal to achieve global convergence. For example, each sizing decision drives buffering additions, and each buffering addition/removal guides the next sizing choices. In this way, buffers are added gradually and gate sizes adapt smoothly to design restructuring. This approach also increases the modularity of the proposed approach: any optimization method that can be applied incrementally to the design can be added or removed from the set available optimization algorithms.

In order to allow for the smooth integration of the various design optimization heuristics and to facilitate the learning process of the MAB algorithms, each design heuristic operates either for a reduced number of iterations, or on a small set of critical paths. In this way, the netlist restructuring achieved by each heuristic is not disruptive to the overall optimization process. If each algorithm run to completion then switching to a new heuristic would incur significant runtime overhead since each new algorithm would need many iterations to reconverge to the new solution after the "disruption" caused by the previously applied heuristic.

A. Gate-sizing Heuristics

Gate sizing is a fundamental step toward achieving timing closure and minimizing the power consumption of a design. It refers to the determination of the widths of the transistors inside a logic gate. Wider transistors imply a faster charging and discharging of the output capacitance. As a drawback, larger transistors increase the input capacitances and, thus, the load, delays, and transition times at the upstream gates.

Many efficient gate-sizing optimization algorithms have been developed over the past decades, starting from simple sensitivity-based heuristics [9] and linear-programming-based solutions [10] to more elaborate optimization methods that are based either on analytical delay models [11] or on employing incremental static timing analysis during the optimization [12]. Lagrange-based relaxation optimizers that cover discrete gate sizes and multiple threshold voltages have been instrumental in achieving high QoRs with fast runtimes [13], [14], [15].

In this work, we employ a modified version of the Lagrange Relaxation-based (LR) engine developed in [15], covering both late and early design constraints [13], [14]. The optimization problem (1) tries to minimize leakage power while respecting all delay constraints and not introducing any maximum-capacitance and maximum-slew violations:

$$\begin{aligned} &\text{minimize } \sum_{gates} leakage \\ &\text{subject to } a_i^L + d_{i\to j}^L \le a_j^L, \forall \text{timing arc } i \to j \\ &\qquad a_i^E + d_{i\to j}^E \ge a_j^E, \\ &\qquad a_k^L \le r_k^L \ , \ a_k^E \ge r_k^E, \forall \text{output } k \end{aligned} \tag{1}$$

$d_{i,j}$ delay is the sum of the wire and the cell delay from the output pin of cell i to the output pin of cell j, while a_k and r_k are the arrival time and the required arrival time (late or early) at node k. The exponents L and E refer to late and early timing information respectively [16].

After introducing Lagrange multipliers to relax the inequality constraints, we end up with the following unconstrained optimization problem:

$$\begin{aligned} &\text{minimize } \sum_{gates} leakage + \\ &\sum_{i\to j} \lambda_{i\to j}^L \left(a_i^L + d_{i\to j}^L - a_j^L\right) + \sum_k \lambda_k^L \left(a_k^L - r_k^L\right) + \\ &\sum_{i\to j} \lambda_{i\to j}^E \left(a_j^E - d_{i\to j}^E - a_i^E\right) + \sum_k \lambda_k^E \left(r_k^E - a_k^E\right) \end{aligned} \tag{2}$$

The Karush–Kuhn–Tucker (KKT) optimality conditions on (2), impose that the Lagrange multipliers must obey network flow conservation constraints:

$$\sum_{u\in fanin(i)} \lambda_{u\to i}^L = \sum_{v\in fanout(i)} \lambda_{i\to v}^L \text{ and } \sum_{u\in fanin(i)} \lambda_{u\to i}^E = \sum_{v\in fanout(i)} \lambda_{i\to v}^E, \forall \text{node } i$$

Otherwise,the arrival time variables a_j are unrestricted. Applying the flow conservation of the lagrange multipliers to (2), allows us to simplify the problem as follows:

$$\begin{aligned} &\text{minimize } \sum_{gates} leakage + \\ &\sum_{i\to j} \lambda_{i\to j}^L d_{i\to j}^L + \sum_{i\to j} \lambda_{i\to j}^E (-d_{i\to j}^E) \end{aligned} \tag{3}$$

Problem (3) is solved by optimal local resizing of one gate at a time, assuming all the other gates are fixed. Gates are traversed in forward topological order from the Primary Inputs (PIs) to the Primary Outputs (POs). The optimal size selected for the gate is the one that minimizes the local delay cost $\sum \lambda_{i\to j}^L d_{i\to j}^L + \lambda_{i\to j}^E (-d_{i\to j}^E)$ of all the gates that participate in the immediate fan-in and fan-out of the cell under examination. In each iteration, the values of the Lagrange multipliers declare the criticality of the corresponding timing arcs and gates. Once a gate is resized, local incremental timing update takes place to locally update the arrival times and the timing slacks of neighboring nodes.

The LR gate sizer converges after many iterations to an optimized solution, where the gate sizes are updated one after the other until no TNS, nor power gains are observed. On the contrary, in this work, we split the multi-iteration LR optimizer to smaller independent heuristics that run for one or two iterations, and possibly to a smaller set of timing-critical gates, thus creating an effectiveness-vs-runtime tradeoff.

LR gate sizing – one iteration – examine all cells (LR1): The optimization loop of the LR gate sizing engine is applied only for one iteration including cell-size and threshold-voltage selection. If TNS is not closed yet, timing optimizations are sought. Once timing constraints are satisfied, LR opts for power reduction.

LR gate sizing – two iterations – examine all cells (LR2): The same as the previous heuristic with the only difference being that LR optimization evolves for two iterations. It trades off increased runtime with better QoR.

LR gate sizing – one iteration – examine only 1K timing-critical cells (LR1C): This heuristic represents a reduced version of the first one. It applies the same one-loop LR optimization, but it touches only the 1K most timing-critical cells. In this way, optimization finishes earlier, with slightly worse QoR.

B. Buffering Heuristics

Buffering optimizations are the Swiss army knife of the optimization algorithms, since they can be efficiently applied for various design targets [17], [18]. In this work, we utilize a small representative set of four buffer-insertion algorithms that target both late and early timing violations, and they are applied both to the data and the clock nets of the design. Each one of the buffer addition steps (buffer removal may be effective in certain circumstances, but it is not used in this work) is applied to a small number of critical paths per trial, in order to (a) keep runtime under control, and (b) allow buffer insertion to be smoothly integrated with gate sizing, which is applied in the subsequent trials.

Late buffering optimization (LB): Add increasingly larger buffer sizes next to the driver of a large-capacitance net until the ratio of the output load to the input load of each gate added locally is approximately 4 (from logical effort [19]).

Early Buffering at the endpoints (EB) : Add buffer with an input capacitance at least as large as the endpoint capacitance. Ensures that extra delay is always added, since the delay of the driving gate remains either the same or it is slightly increased.

Clock buffering insertion (CB): Insert an additional local-clock buffer on the clock pin of a flip-flop if we need to slow down the clock signal for this register, i.e., the D-pin late

TABLE I
THE RUNTIME SCALING FACTOR USED FOR TUNING THE REWARD ACCORDING TO THE RUNTIME SPENT BY EACH HEURISTIC TO ACHIEVE IT.

Optimization method	r_a
LB–Late buffering	1.00
CPI–Critical Path Isolation	1.00
EB–Early buffering	1.00
CB–Clock buffering	1.00
LR1C–LR 1 Iteration for critical cells	0.85
LR1–LR 1 Iteration for all cells	0.75
LR2–LR 2 Iterations for all cells	0.60

slack is more critical than the Q-pin late slack, or the Q-pin early slack is more critical than the D-pin early slack. We do not insert buffers if both sides are not critical.

Buffering for critical path isolation (CPI): Reduce the input capacitance of the non-critical branches of a net by adding buffers at the non-critical sinks, with an input capacitance smaller than the capacitance at each sink. In this way, the driving gate sees a smaller output load, which it decreases.

IV. REWARD FUNCTION

The reward function should demonstrate the effect of the chosen optimization method on the performance metrics that characterize the circuit's behavior. The reward given to algorithm a at trial t, i.e., $Q_t(a)$, quantifies the improvement achieved by algorithm a in TNS and leakage power:

$$Q_t(a) = r_a \left(\delta \cdot qTNS + (1-\delta) \cdot qPower\right)$$

The parameters $qTNS$ and $qPower$ are bound between 0 and 1, and they represent how efficient algorithm a was in reducing TNS and leakage power, respectively. Factor δ defines the importance of TNS optimization compared to leakage-power optimization, and can be changed during the flow depending on the design's current state. In this work, we target timing closure in the first optimization rounds and, thus, $\delta = 0.8$, while, once timing closure is achieved, power reduction is prioritized with $\delta = 0.2$.

Parameter r_a is a scaling factor that depends solely on the runtime complexity of the optimization algorithm a. Each optimization method has a different runtime complexity. We want to include this in our reward function, so that the MAB algorithm is able to effectively distinguish between two methods that have a similar QoR impact, but different runtime needs. In this way, we penalize the reward received by slow methods with low values for $r(a)$. The values of r_a used for each method are statically chosen and are listed in Table I.

A. TNS reward

$qTNS$ quantifies by how much TNS was improved in this trial relative to how much leakage power was increased. It is calculated separately for early and late timing modes $qTNS = 0.5 \cdot qTNS_{early} + 0.5 \cdot qTNS_{late}$, according to the following rules:

R1–Unproductive optimization: If TNS was degraded or remained unchanged, the algorithm receives $qTNS = 0$.

R2–Always productive: If both TNS and leakage power were reduced $qTNS = 1$, i.e., the algorithm receives the maximum reward,

R3–Tradeoff: If TNS was improved and the leakage power was increased, then

$$qTNS = \min\left(\frac{\Delta TNS}{\gamma\ \Delta P}, 1\right)$$

The $qTNS$ reward quantifies the percentage of TNS improvement over the percentage of power increase. This value can be larger than 1, so we apply a check and set it to 1 if it exceeds it. ΔTNS and ΔP represent the percentage of TNS and power change (between trials t and $t-1$) after the application of the selected optimization algorithm, i.e.,

$$\Delta TNS = \frac{TNS_t - TNS_{t-1}}{TNS_{t-1}} \text{ and } \Delta P = \frac{Power_t - Power_{t-1}}{Power_{t-1}}$$

When TNS_{t-1} is close to 0, it means that timing constraints are almost met. In this case, trying to calculate the relative TNS increase would lead to false/misleading conclusions. Due to the TNS_{t-1} term being the denominator, as its value approaches 0, any increase in TNS_t away from 0 would amount to a near-infinite relative increase, even though the absolute increase is – in reality – very small. Hence, any increase from a small TNS value would probably end up being artificially amplified, thereby derailing the convergence of the optimization algorithm.For example, let us assume that $TNS_{t-1} = -2$, and, after applying the next method, $TNS_t = -6$. The relative degradation would equal the huge value of 200%, while the real degradation is a minor setback compared to the original TNS at the beginning of the optimization process. To combat this artifact in the calculation of ΔTNS, whenever the value of TNS_{t-1} falls below a threshold value (arbitrarily chosen to be 10% of the clock period in our experiments), then the TNS degradation is quantified differently, as $\Delta TNS = TNS_{t-1}/T_{clock}$.

Variable γ defines what $TNS/power$ tradeoff we consider satisfactory. For example, let us assume that the TNS improvement is 40%, for a power increase of 20%. For $\gamma = 1$, the ratio of TNS change to power change will be equal to 2 $(0.4/0.2)$, thus setting $qTNS = 1$. In the case that $\gamma = 4$, TNS reward will be equal to $0.5 = \min\left(\frac{0.4}{0.2 \cdot 4}, 1\right)$. The value of γ should be neither too large, nor too small. A large value for γ will result in most rewards receiving very small values, whereas a small value of γ will result in most rewards being set to 1. In the experimental results presented in this paper, we assumed $\gamma = 4$.

B. Power Reward

The reward with respect to leakage power, $qPower$ is calculated in a similar manner:

R1–Unproductive optimization: If leakage power was increased or remains unchanged: $qPower = 0$.

R2–Always productive: If leakage power was reduced and the TNS was improved: $qPower = 1$. In this way, the algorithm achieved a truly productive step, since it achieved both a reduction in power and a timing optimization in the same step.

TABLE II
COMPARING THE QUALITY OF THE AUTONOMOUS MAB ORCHESTRATION TO THE BEST PUBLISHED RESULT FOR THE ISPD12 BENCHMARK SET

Benchmark	#Cells	Leakage (mW)		Optimization algorithms						
		[13]	MAB	CPI	LB	EB	LR1	LR1C	LR2	#Trials
DMA_slow	25300	132	134	8	6	5	10	9	10	48
DMA_fast		238	248	11	9	8	14	11	10	63
pci_bridge3_slow	33203	96	96	10	7	6	10	9	12	54
pci_bridge3_fast		136	138	6	6	4	11	5	11	43
des_perf_slow	111228	570	601	9	6	4	11	8	5	43
des_perf_fast		1395	1511	8	7	5	11	12	12	55
vga_lcd_slow	164890	328	335	6	5	4	6	8	4	33
vga_lcd_fast		412	430	10	6	6	10	10	9	51
b19_slow	219268	564	660	4	2	2	4	3	3	18
b19_fast		717	769	6	3	3	5	5	4	26
leon3mp_slow	649190	1334	1340	6	6	3	5	5	5	30
leon3mp_fast		1443	1530	6	7	4	9	6	9	41
netcard_slow	958780	1763	1755	4	3	2	4	2	3	18
netcard_fast		1841	1884	4	3	3	5	4	5	24
Average		783.5	816.5	7.0	5.4	4.2	8.2	6.9	7.3	39.1

R3–Tradeoff: If the leakage power was reduced and the TNS was degraded, $qPower$ quantifies the relative percentage of power decrease relative to the TNS increase:

$$qPower = \min\left(\frac{\Delta P}{\gamma\ \Delta TNS}, 1\right).$$

C. Reward expansion

The reward $Q_t(a)$ given to algorithm a at trial t is always in the range [0,1]. Once the MAB-based optimization has played for many trials, we may observe an algorithm achieving only a marginal improvement compared to the improvements achieved during the first optimization rounds. Therefore, the reward given to an algorithm is very low and the learning algorithm cannot effectively distinguish the efficient algorithms. To avoid this, we renormalize the reward function within the range [0,1], with the goal to enhance small rewards and compress very large ones: $Q_t^{\text{new}}(a) = \log(1-\frac{1}{b})\log(1-\frac{Q_t(a)}{b})$. Assuming a small value for $b = 0.05$, we are able to boost the small rewards observed. This is useful near the end of the simulation runs, where hard-to-verify bins remain uncovered.

V. EXPERIMENTAL RESULTS

The proposed design optimization flow has been implemented in C++ using the open-source RSyn physical design framework [20]. RSyn provides all necessary supporting functions for netlist traversal and manipulation, as well as cell re-powering and incremental timing analysis needed by the proposed flow. The new method is evaluated using two benchmarks sets; namely, the ISPD 2012 gate-sizing benchmarks [7] and the TAU 2019 benchmarks for design optimization [8]. The final results are validated with OpenTimer [21], which is the reference timer used for evaluation in both above-mentioned contests.

In the case of the ISPD 2012 benchmark set, we compare against the best results achieved so far in the open literature [13] for this benchmark set. The best results are derived from a Lagrange Relaxation-based optimization that runs for 120 iterations, together with final timing and power-recovery heuristics. The proposed MAB-based framework successfully approaches the best published results with marginal differences in less iterations, as shown in Table II, by using a fully autonomous optimization and with no human in the loop. Note that Table II reports only the achieved leakage power, since, in all cases, initial timing violations are closed without introducing any maximum-capacitance or maximum-slew violations. In the case of the ISPD 2012 benchmarks, clock-tree buffering is not applied, since the included library does not include any clock buffers. These results are of paramount significance and unequivocally exemplify the strength of a MAB-based methodology: it paves the way for fully autonomous, self-driving, "no-human-in-the-loop" automation flows *without compromising (trading away) PPA quality* [22].

The MAB methodology chooses online which optimization algorithm to play in each round and records its reward. The reward that corresponds to the algorithms not played so far remains hidden. Therefore, the only way to learn this information is to repeatedly try all algorithms (exploration). In parallel, whenever the MAB process pulls a bad algorithm, it suffers some regret. The MAB methodology should reduce the regret by repeatedly pulling the best algorithm (exploitation)

The number of times that each algorithm has been selected for each benchmark is also shown in Table II. On average, the heuristic with the most plays is LR1, followed by LR2, and CPI. As the optimization progresses, the leakage power reduction becomes smaller after each iteration. This means that the leakage power benefit of LR1 is more than half of the benefit of LR2. Due to the runtime scaling factor (r_a), UCB (the driving force behind MAB) ends up favoring the LR1. LR for critical cells is mostly beneficial in the early optimization stages, but it can also contribute in minimizing the small timing violations that might appear during the leakage-power optimization. CPI is the most efficient buffer insertion algorithm for these designs, as is shown by its high pick rate. The other two algorithms have minimal effect: (1) LB, because of the absence of very high fanout nets, and (2) EB, because of the absence of early timing violations in this benchmark set. Consequently, these two algorithms have the lowest pick rate.

The same experiments were repeated on the benchmarks of the TAU 2019 contest, using the typical library corner available. The obtained results are depicted in Table III. It is evident that in all cases, timing constraints are met for both early and late timing while leakage power is significantly reduced. These benchmarks have a very high amount of hold violations. Gate sizing has a limited ability to fix them, as

TABLE III
INITIAL AND FINAL TIMING AND POWER OF THE TAU 2019 BENCHMARK SET UNDER TYPICAL CORNER. THE ALGORITHM RECOMMENDATIONS MADE BY THE MAB POLICY.

Benchmark	#Cells	Initial					MAB				
		Late (ps)		Early (ps)		Leakage(uW)	Late (ps)		Early (ps)		Leakage (uW)
		WNS	TNS	WNS	TNS		WNS	TNS	WNS	TNS	
s1196	642	0	0	0	0	30	0	0	0	0	12.8
systemcdes	3441	0	0	346	5809	181	0	0	0	0	79
usb_funct	15743	0	0	984	244911	788	0	0	0	0	511
vga_lcd	139529	1499	14889000	2593	53407000	6002	0	0	0	0	3840

Benchmark	Optimization algorithms						
	LB	EB	CPI	CB	LR1C	LR1	LR2
s1196	2	1	2	1	1	1	2
systemcdes	2	4	2	2	4	2	4
usb_funct	5	8	5	11	8	9	11
vga_lcd	4	8	4	11	8	14	12

opposed to early buffer insertion, which inserts buffers directly at the violating endpoints, being able to virtually solve any hold violations. Thus, it is not surprising that the MAB algorithm picks this method the most. All gate-sizing methods have a similar pick rate, lower than the EB and higher than the LB. Since the designs have easy-to-fix late violations for this corner, LB is picked the least.

VI. CONCLUSIONS

The criticality of timing-driven design optimization during physical synthesis is accentuated as modern hardware designs become increasingly complex and transistor and wire scaling are no longer very helpful. In order to contain the runtime and increase the effectiveness of timing-driven design optimization, scalable automation processes are needed to guide the optimization flow.

In this context, this work leverages reinforcement learning to automatically and autonomously apply optimization heuristics to unknown designs and improve their PPA metrics. The proposed MAB-based framework operates on-line, without any previous training, or knowledge of the design, or the optimization methods used. Yet, it achieves PPA metrics that approach the best published results so far. Most importantly, the introduced approach allows for a modular construction of a customized optimization flow, whereby optimization algorithms are added, or removed (even on-line), according to their effectiveness on a per-design basis.

ACKNOWLEDGMENTS

Dimitrios Mangiras is supported by the Onassis Foundation - Scholarship ID: G ZO 014-1/2018-2019. This research has been supported by a research grant from Mentor, a Siemens Business to DUTH.

REFERENCES

[1] N. D. MacDonald, "Timing closure in deep submicron designs," in *Design Automation Conference (DAC)*, 2010.
[2] J. White, *Bandit Algorithms for Website Optimization*. O'Reilly Media, 2012.
[3] T. Lattimore and C. Szepesvari, *Bandit Algorithms*, 2018. [Online]. Available: http://banditalgs.com
[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proc. of the Intern. Conf. on Parallel Architectures and Compilation (PACT)*, 2014, pp. 303–316.
[5] C. Xu, G. Liu, R. Zhao, S. Yang, G. Luo, and Z. Zhang, "A parallel bandit-based approach for autotuning fpga compilation," in *Proc. of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 157–166.
[6] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, May 2002.
[7] M. Ozdal and et al., "The ISPD-2012 discrete cell sizing contest and benchmark suite," in *Intern. Symp. on Physical Design (ISPD)*, 2012.
[8] "Tau 2019 contest – design optimization," 2019. [Online]. Available: https://sites.google.com/view/tau-contest-2019/
[9] J. Hu and et al., "Sensitivity-guided metaheuristics for accurate discrete gate sizing," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2012, p. 233–239.
[10] D. G. Chinnery and K. Keutzer, "Linear programming for sizing, vth and vdd assignment," in *Proc. of the Intern. Symp. on Low Power Electronics and Design (ISLPED)*, 2005, pp. 149–154.
[11] C.-P. Chen, C. C. N. Chu, and D. F. Wong, "Fast and exact simultaneous gate and wire sizing by lagrangian relaxation," *IEEE Trans. on CAD*, vol. 18, no. 7, pp. 1014–1025, July 1999.
[12] A. B. Kahng, S. Kang, H. Lee, I. L. Markov, and P. Thapar, "High-performance gate sizing with a signoff timer," in *International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 450–457.
[13] G. Flach and et al., "Effective method for simultaneous gate sizing and vth assignment using lagrangian relaxation," *IEEE Trans. on CAD*, vol. 33, no. 4, pp. 546–557, April 2014.
[14] M. M. Ozdal, S. Burns, and J. Hu, "Algorithms for gate sizing and device parameter selection for high-performance designs," *IEEE Trans. on CAD*, vol. 31, no. 10, pp. 1558–1571, October 2012.
[15] A. Sharma, D. Chinnery, S. Bhardwaj, and C. Chu, "Fast lagrangian relaxation based gate sizing using multi-threading," in *IEEE/ACM Inter. Conf. on Computer-Aided Design*, 2015, pp. 426–433.
[16] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.
[17] C. B. Yanbin Jiang, S.S. Sapatnekar and J. Kim, "Interleaving buffer insertion and transistor sizing into a single optimization," *IEEE Trans. on VLSI Systems*, vol. 6, no. 4, pp. 625–633, 1998.
[18] and M. D. F. Wong, I. Nedelchev, S. Bhardwaj, and V. Parkhe, "On timing closure: Buffer insertion for hold-violation removal," in *Design Automation Conference (DAC)*, 2014.
[19] B. S. Ivan Sutherland and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann, 1999.
[20] G. Flach, M. Fogaça, J. Monteiro, M. Johann, and R. Reis, "Rsyn: An extensible physical synthesis framework," in *Intern. Symp. on Physical Design (ISPD)*, 2017, pp. 33–40.
[21] T. W. Huang and M. D. F. Wong, "Opentimer: A high-performance timing analysis tool," in *Intern. Conf. on Computer-Aided Design (ICCAD)*, 2015, pp. 895–902.
[22] S. Sadasivam, Z. Chen, J. Lee, and R. Jain, "Efficient reinforcement learning for automating human decision-making in soc design," in *Design Automation Conference (DAC)*, 2018.