HYPERHEURIST: A Simulated Annealing-Based Control Framework for LLM-Driven Code Generation in Optimized Hardware Design

Shiva Ahir
Department of ECE
Stony Brook University
Stony Brook, NY
shiva.ahir@stonybrook.edu

Prajna Bhat
Department of ECE
Stony Brook University
Stony Brook, NY
prajna.bhat@stonybrook.edu

Alex Doboli
Department of ECE
Stony Brook University
Stony Brook, NY
alex.doboli@stonybrook.edu

***Abstract*—Large Language Models (LLMs) have shown promising progress for generating Register Transfer Level (RTL) hardware designs, largely because they can rapidly propose alternative architectural realizations. However, single-shot LLM generation struggles to consistently produce designs that are both functionally correct and power-efficient. This paper proposes HYPERHEURIST, a simulated-annealing–based control framework that treats LLM-generated RTL as intermediate candidates rather than final designs. The suggested system not only focuses on functionality correctness but also on Power-Performance-Area (PPA) optimization. In the first phase, RTL candidates are filtered through compilation, structural checks, and simulation to identify functionally valid designs. PPA optimization is restricted to RTL designs that have already passed compilation and simulation. Evaluated across eight RTL benchmarks, this staged approach yields more stable and repeatable optimization behavior than single-pass LLM-generated RTL.**

***Index Terms*—Large Language Models, Automated Code Generation, Heuristic Optimization, RTL Synthesis, Functional Correctness, Design Space Exploration**

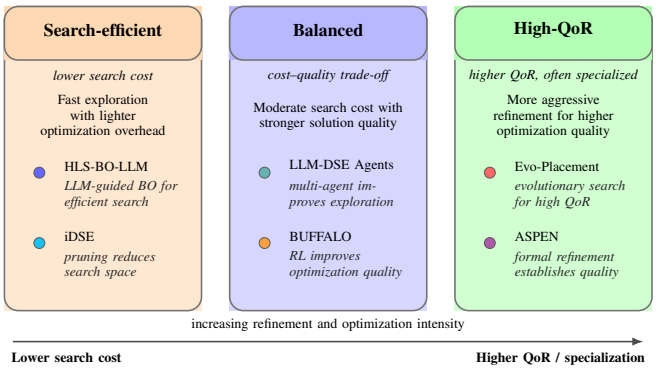

Fig. 1: Categorization of representative LLM-based heuristic-generation frameworks into search-efficient, balanced, and high-QoR regimes. Each method is annotated by its core mechanism, showing how different approaches trade off search cost, generality, and optimization quality.

I. Introduction

Optimization in hardware design has traditionally relied on fixed rules, manually tuned heuristics, and scripts from experienced verification engineers. As designs grow larger and constraints become tighter, these approaches increasingly struggle to scale because they require constant retuning and a lot of domain expertise. Recent work shows that LLMs can change this pattern by instead of hard coding heuristics, LLMs can generate and adapt them based on feedback from synthesis, simulation, and quality metrics [1]–[3]. Prior work [1], [4]–[6] shows this idea across multiple stages of the design flow, which includes high-level synthesis, physical design, placement, buffering, and rewriting. LLMs move beyond simple suggestions and begin to guide search, balance exploration and refinement, and react to performance feedback, such as timing, area, and power.

In spite of this progress, existing approaches arguably remain fragmented [2], [5], [6], [11]. Most are limited to a single design stage and use custom prompting or agent structures that do not transfer well across problems. Feedback mechanisms also vary widely, and durability across designs, tools, and technology settings is still limited. This paper introduces HYPERHEURIST as a unifying framework to address these gaps. HYPERHEURIST generates SystemVerilog code for digital hardware by treating heuristic design itself as a learnable process and applies it to RTL hardware optimization. It combines parallel heuristic generation, refinement that is critique-driven, and Simulated Annealing [7] with a simpler targeted pipeline for focused optimization. A contextual mechanism selects the strategy that best fits the problem and feedback, moving toward a more general and adaptive model of heuristic-driven EDA.

The paper has the following structure. Section II presents related work and motivation. Section III presents methodology, Section IV describes experimental setup, and Section V discusses results.

II. Related Work and Motivation

Recent research has explored LLMs as active components in optimization workflows across high-level synthesis (HLS), RTL generation, and physical design. Rather than serving as passive code generators, related work positions LLMs as heuristic generators that propose, refine, or adapt optimization strategies under tool feedback [1]–[3]. Figure 1 categorizes

TABLE I: Operational and Qualitative Comparison of Six LLM-Based Heuristic-Generation Frameworks

Framework	Design Stage	Core Method	Heuristic Target	Search Cost	Generality	Strengths	Limitations
HLS-BO-LLM [1]	HLS DSE	LLM + Bayesian opt.	Directive configs (pragmas)	Low–Medium	Tool-specific	Strong QoR vs. random/grid; interpretable directives	Dependent on surrogate quality and HLS API
iDSE [2]	HLS DSE	LLM-guided pruning + seeding	Seeds + filters for directives	Low	Moderate	Large reduction in explored designs; convergent/divergent reasoning	Requires tuned prompting and HLS feedback loops
LLM-DSE Agents [3]	HLS / accelerator params	Multi-agent LLM search	Parameter tree search policy	Medium	Cross-tool potential	Flexible agent roles; good under budgeted evaluations	Coordination overhead; complex to debug
Evo-Placement [4]	Global placement	LLM-generated alg. variants	Initialization + update rules	Medium–High	Node-specific	Non-trivial HPWL improvements; discovers novel optimizers	Training/evolution cost; integration with legacy placers
ASPEN [5]	RTL datapath	LLM + e-graphs + formal checks	Rewrite rules and extraction	Medium	Technology-aware	Combines correctness guarantees with PPA gains	E-graph infra and solver overhead; corpus-dependent
BUFFALO [6]	Buffer-tree / CTS	LLM + GRPO RL	Tree topology and sizing	Low (inference-time)	Industrial-scale	Large timing improvements; orders-of-magnitude runtime gains	Heavy offline training; task-specialized sequence model

existing LLM-based hardware generation approaches into three regimes, namely search-efficient, balanced, and high-QoR, illustrating the trade-offs between optimization quality, search effort, and reliance on tool-driven feedback. Table I summarizes the characteristics of related work.

The first category focuses on HLS-driven design space exploration, where heuristics operate over pragma or parameter spaces. Yao et al. [1] integrate an LLM into a Bayesian optimization loop to guide pragma selection. Li et al. [2] extend this idea in iDSE by using the LLM to prune directive spaces, seed promising configurations, and adapt exploration based on observed QoR trends. Wang et al. [3] propose a multi-agent framework in which router, specialist, and critique agents collaboratively traverse parameter trees, with the LLM steering refinement using synthesis feedback. This work showcases that LLMs can reason quite effectively over structured directive spaces when coupled with numerical evaluation signals.

More recent efforts extend LLM-guided heuristics into RTL and physical design, where models generate structured optimization artifacts rather than simple parameter selections. LLMs synthesize program-level constructs such as placement update rules, rewrite sequences, or structural layouts, which are evaluated using synthesis, equivalence checking, and Power-Performance-Area (PPA) feedback. Results show that LLMs can interact meaningfully with downstream EDA tools when constrained to emit executable, tool-evaluable artifacts.

However, prior frameworks also expose persistent limitations [5], [6], [8]. Many prior approaches couple functional correctness and QoR optimization within a single search loop, wasting effort on unstable designs. Others use weakly regulated exploration, leading to oscillation, premature convergence, and structural drift. In contrast, HYPERHEURIST introduces phase-decoupled correctness gating, stabilizing search by separating correctness discovery from PPA refinement. It models RTL implementations as explicit search states, leverages tool-in-the-loop feedback, and regulates exploration using Simulated Annealing with specialized generation, mutation, and critique roles.

III. The HYPERHEURIST Framework

This paper proposes *HYPERHEURIST*, a two-phase heuristic generation framework for RTL code that prioritizes functional correctness before discovering power, performance, and area (PPA) optimization. The central design principle is that correctness and optimization impose fundamentally different search dynamics and should therefore be handled by distinct but coordinated adaptive strategies. By decoupling these objectives, HYPERHEURIST avoids expending expensive optimization effort on invalid designs while allowing aggressive exploration once correctness is established.

A. Framework Architecture

Figure 2 illustrates the conceptual architecture of the proposed HYPERHEURIST framework. The framework executes two adaptive strategies sequentially: a correctness-driven search phase followed by a PPA-oriented optimization phase. We separate these phases because early PPA optimization on partially correct RTL repeatedly produced misleading results in our experiments.

During execution, candidate RTL designs are evaluated using Synopsys VCS simulation [16] and synthesis feedback. The system records the best candidate based on compilation success, simulation pass/fail status, and synthesized area. Both phases execute feedback-driven search loops, but differ in objective: Phase 1 prioritizes functional validity, while Phase 2 refines power, performance, and area under correctness-preserving constraints.

1) Adaptive Strategy 1: Correctness-Driven Search: This strategy prioritizes discovering functionally valid RTL designs before any optimization is performed. A heuristic generator

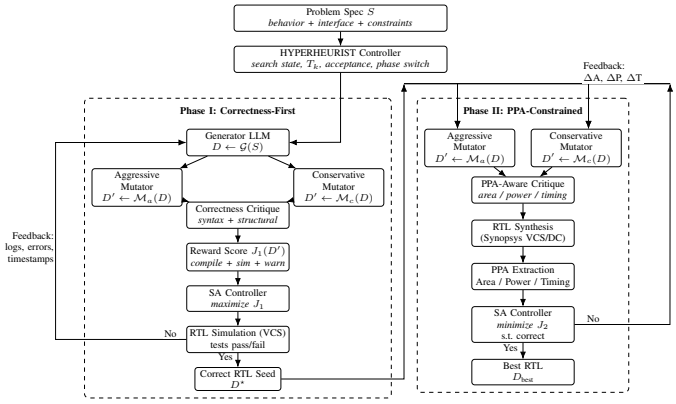

Fig. 2: Two-phase HYPERHEURIST framework. Phase I discovers functionally correct RTL seeds using tool-in-the-loop simulation feedback. Phase II refines the seed via synthesis-driven PPA optimization under a strict correctness constraint.

proposes diverse RTL candidates from the problem specification, emphasizing architectural coverage rather than quality. These candidates are refined through two mutation pathways: a conservative mutator that applies structure-preserving fixes to stabilize existing designs, and an aggressive mutator that introduces larger architectural changes to explore alternative datapaths or control structures.

Each candidate is evaluated by a structural critique that verifies key correctness properties, including reset behavior, assignment discipline, FSM completeness, and pipeline consistency. Compilation, simulation, and structural checks are combined into a scalar correctness reward, which serves as the energy function for a SA controller. This controller probabilistically accepts candidates to balance exploration with convergence toward robust designs. Evaluation feedback is reused to guide subsequent mutations, while the best correctness-preserving candidates are continuously tracked. The output of this phase is a small set of RTL implementations that are functionally verified and structurally stable.

2) Adaptive Strategy 2: PPA-Oriented Optimization: This strategy optimizes power, performance, and area while strictly preserving functional correctness. Mutation and critique retain the same structural roles as in the correctness-driven phase, but mutations are now biased toward synthesis-friendly refinements. Functional correctness is enforced as a hard constraint, and any violating candidate is immediately discarded.

Correct candidates are synthesized to obtain timing, area, and power metrics, which are combined into a composite PPA objective. A SA controller regulates candidate acceptance using this objective, enabling controlled exploration of PPA trade-offs without compromising correctness.

Synthesis feedback is incorporated iteratively, and the framework continuously tracks the best-performing design that remains functionally equivalent to the verified seeds from Phase 1. The output of this phase is a final RTL implementation optimized for PPA under strict correctness constraints.

TABLE II: Eight RTL problem statements adopted from the RTLLLM [13] paper

Design	Description
`serial2parallel_8`	1-bit serial input and output data after receiving 6 inputs
`alu4`	Arithmetic logic unit operating on 4-bit inputs
`counter_0_12`	Counter module counts from 0 to 12
`traffic_light`	Traffic light system with three colors and pedestrian button
`freq_div`	Frequency divider for 100 MHz input clock producing lower-frequency outputs
`johnson_counter`	4-bit Johnson counter with specific cyclic state sequence
`mux2_sync`	Multi-bit synchronous multiplexer
`parallel2serial`	Convert 4 input bits into a single serial output bit

B. Prompts and Experimental Setup

1) Benchmarks and Problem Statements: We evaluated HYPERHEURIST using **8 RTL problem statements** adopted from the *RTLLLM* benchmark set [13]. These problem statements are summarized in Table II. Each benchmark provides a compact natural-language specification and an expected RTL interface, enabling a consistent comparison across baseline prompting and our multi-pipeline hyper-heuristic search.

2) Multi-Pipeline Prompting (Four LLM Roles): For each problem statement, HYPERHEURIST initialized a candidate pool using four distinct LLM pipelines (roles). The goal was to induce complementary behaviors (diverse exploration vs. safe refinement) while maintaining a consistent output contract (synthesizable RTL).

- **Pipeline A: Generator** produces a clean, standard RTL implementation targeting correctness.
- **Pipeline B: Conservative Mutator** makes minimal, low-risk edits (small rewrites, reset fixes, interface consistency).
- **Pipeline C: Critique** reviews the candidate RTL and emits actionable defect hypotheses and patch suggestions.
- **Pipeline D: Aggressive Mutator** performs larger transformations (state encoding changes, pipelining, refactoring) to escape local minima.

3) Common System Contract: All pipelines share a strict system-level contract to ensure compilation and synthesis compatibility.

```
SYSTEM_BASE:
You are an expert RTL engineer.
Return ONLY synthesizable SystemVerilog code
(no markdown, no explanation).
Keep module name and ports exactly as
specified.
```

4) Role-Specific Prompt Templates: Each pipeline receives the same benchmark specification but is conditioned using a role-specific prompt template. Curly-brace fields denote runtime substitution during execution.

GENERATOR_PROMPT:

```
You are a RTL engineer.
TASK: Parameterizable Johnson counter (W=8):
if ce, shift and set q[0]<=~q[W-1]. Add
sync active-high reset to a known state +
illegal-state recovery to reset state.
CONSTRAINTS: Synthesizable SystemVerilog;
always_ff/always_comb separated; fully
synchronous; all regs reset.
OUTPUT: Synthesizable SystemVerilog only.
```

CONSERVATIVE_MUTATOR_PROMPT:

```
REF: {rtl}
TASK: Fix correctness/synth/lint issues
without changing architecture; keep
reset/enable behavior consistent; confirm
illegal-state recovery.
CONSTRAINTS: Keep ports/params and pipeline
depth unchanged; synchronous reset.
OUTPUT: Synthesizable SystemVerilog only.
```

AGGRESSIVE_MUTATOR_PROMPT:

```
REF: {rtl}
TASK: Explore alternate micro-architecture
(e.g., optional staging/enable pipelining)
while preserving spec and recovery behavior.
CONSTRAINTS: Keep ports/params unchanged;
synchronous reset; synthesizable
SystemVerilog.
OUTPUT: Synthesizable SystemVerilog only.
```

CRITIQUE_PROMPT:

```
SPEC: Johnson counter (W=8) with ce,
inverted-MSB feedback, sync active-high reset,
and illegal-state recovery.
RTL: {rtl}
OUTPUT: JSON only: {syntax, reset, logic,
hazard} in {0.0,0.5,1.0}.
```

5) Tool-in-the-Loop Verification and Feedback: We compiled and simulated each candidate using `Synopsys VCS W-2024.09-SP1_Full64` [16]. If a candidate failed at any stage, the tool-generated error outputs were fed back into the next LLM call to drive targeted repair. The feedback packet included: (i) compile errors, (ii) simulation failures, (iii) warnings, and (iv) timestamps/error identifiers when available.

Compile/Simulation Loop. Given candidate RTL x, we ran:

1) **Compile check:** validate syntax and elaboration.
2) **Simulation check:** execute the benchmark testbench and record pass/fail.
3) **Log feedback:** on failure, extract a compact log slice (first error + context) and return it to the selected pipeline for repair.

HYPERHEURIST uses the same Simulated Annealing (SA) control logic in both phases. The only difference is the *objective function* used to score a design and the *constraints* enforced by the evaluator: (i) Phase 1 optimizes correctness, and (ii) Phase 2 optimizes PPA while preserving correctness. Algorithm 1 presents the unified SA routine.

TABLE III: Syntax and Functional Correctness Comparison Across LLM-Based RTL Generation Frameworks

Design	RTL–LLM [13]				HYPERHEURIST	
	GPT-3.5 + SP		GPT-4		GPT-4.0	
	Syn	Func	Syn	Func	Syn	Func
serial2parallel_8	✓	✓	✓	✓	✓	✓
alu4	✗	✗	✓	✗	✓	✓
counter_0_12	✓	✓	✓	✓	✓	✓
traffic_light	✓	✗	✓	✓	✗	✗
freq_div	✓	✗	✓	✗	✓	✓
johnson_counter	✓	✓	✓	✓	✓	✓
mux2_sync	✓	✗	✓	✓	✓	✓
parallel2serial	✓	✗	✓	✗	✓	✗

Algorithm 1 Unified Simulated Annealing for HYPERHEURIST (Phase 1 & Phase 2)

Require: Benchmark specification S; initial design D_0; max iterations K; initial temperature T_0; cooling factor $\alpha \in (0,1)$; minimum temperature $T_{\min}$; mode schedule $\textsc{Mode}(k) \in \{\text{P1}, \text{P2}\}$.
Ensure: Best design found D_{best} and score J_{best}.

$D \leftarrow D_0$
$(J, \textsc{Logs}) \leftarrow \textsc{Evaluate}(D, S, \textsc{Mode}(0))$
$D_{\text{best}} \leftarrow D, \quad J_{\text{best}} \leftarrow J$
$T \leftarrow T_0$
for $k = 1$ to K **do**
  if $T < T_{\min}$ **then**
    break
  end if
  $m \leftarrow \textsc{Mode}(k)$
  ▷ LLM-guided neighbor generation may use feedback logs
  $D' \leftarrow \textsc{MutateLLM}(D, S, m, \textsc{Logs})$
  $(J', \textsc{Logs}') \leftarrow \textsc{Evaluate}(D', S, m)$
  ▷ Unified SA acceptance on scalar objective J (higher is better)
  $\Delta \leftarrow J' - J$
  if $\Delta \geq 0$ **then**
    $D \leftarrow D', \ J \leftarrow J', \ \textsc{Logs} \leftarrow \textsc{Logs}'$
  else
    accept with probability $p = \exp(\Delta/T)$
    if accepted **then**
      $D \leftarrow D', \ J \leftarrow J', \ \textsc{Logs} \leftarrow \textsc{Logs}'$
    end if
  end if
  if $J > J_{\text{best}}$ **then**
    $D_{\text{best}} \leftarrow D, \quad J_{\text{best}} \leftarrow J$
  end if
  $T \leftarrow \alpha \cdot T$
end for
return $D_{\text{best}}, J_{\text{best}}$

C. Unified Simulated Annealing Procedure for Phase 1 and Phase 2

1) How the Algorithm Works in Both Phases: Both phases share the same algorithmic structure; they differ only in the evaluation objective and acceptance rule.

- **Phase 1 (Correctness SA):** EVALUATE maximizes a correctness score. Candidates with $\Delta \geq 0$ are always ac-

cepted, while worse candidates are accepted with probability $\exp(\Delta/T)$ to escape partially correct RTL plateaus.
- **Phase 2 (PPA SA):** EVALUATE minimizes a PPA cost. Candidates with $\Delta \leq 0$ are always accepted, while degradations are accepted with probability $\exp(-\Delta/T)$ to explore beyond local optima.
- **Mutation and constraints:** MUTATELLM is guided by tool feedback, compilation and simulation logs in Phase 1 [9], [10], and synthesis, timing, and power reports in Phase 2, while Phase 2 mutations strictly preserve functional correctness.
- **Temperature schedule:** High temperature promotes exploration, with gradual cooling enforcing convergence.

IV. EXPERIMENTAL CONTROLS AND REPRODUCIBILITY

All HYPERHEURIST correctness and PPA results (Tables IV–V) were obtained under controlled conditions, while baseline correctness results in Table III are reproduced from prior work [13]. Each framework was evaluated using the same toolchain, constraints, and verification flow so that observed differences reflect only the generation strategy.

a) Controlled tool-in-the-loop evaluation: For each benchmark, RTL candidates were generated automatically and evaluated using a fixed two-stage pipeline:

1) Front-end correctness validation via compilation and simulation using *Synopsys VCS* [16], enforcing consistent testbench execution and deterministic pass/fail outcomes.
2) Back-end PPA evaluation via logic synthesis using *Synopsys Design Compiler* (DC) [16] with a fixed standard-cell library and identical timing constraints across all benchmarks. .

Listing 1: Correct and stable RTL for `freq_div`

```
module freq_div #(parameter DIV = 100)(
  input  logic clk,
  input  logic rst_n,
  output logic clk_out
);
  logic [$clog2(DIV)-1:0] cnt;

  always_ff @(posedge clk) begin
    if (!rst_n) begin
      cnt <= '0;              // Proper reset
      clk_out <= 1'b0;        // Deterministic
          initialization
    end else if (cnt == DIV-1) begin
      cnt <= '0;
      clk_out <= ~clk_out;    // Controlled toggle
    end else begin
      cnt <= cnt + 1'b1;
    end
  end
endmodule
```

This design synthesizes efficiently due to minimal control logic and predictable switching activity and leads to reduced area and power in Table IV. No manual edits, constraint tuning, or post-processing were applied.

Let N denote the total number of generated candidates. For candidate i, define:

$$\mathbb{I}_{\text{vcs}}^{(i)} = \begin{cases} 1 & \text{if VCS compilation and simulation succeed} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Only candidates with $\mathbb{I}_{\text{vcs}}^{(i)} = 1$ were forwarded to synthesis, ensuring that all reported PPA results correspond to functionally valid RTL.

b) PPA objective formulation.: For each validated candidate, DC reports:

$$A^{(i)}, \quad P^{(i)}, \quad T^{(i)}.$$

The final design is selected by minimizing:

$$\mathcal{J}^{(i)} = \alpha \cdot \widehat{A}^{(i)} + \beta \cdot \widehat{P}^{(i)} + \gamma \cdot \widehat{T}^{(i)}, \tag{2}$$

where $\widehat{(\cdot)}$ denotes per-benchmark normalization and α, β, γ are fixed weights. The objective is evaluated over the feasible set:

$$\mathcal{F} = \{i \mid \mathbb{I}_{\text{vcs}}^{(i)} = 1\},$$

and the reported design is $i^\star = \arg\min_{i \in \mathcal{F}} \mathcal{J}^{(i)}$.

c) Representative successful case.: The `freq_div` benchmark illustrates a successful outcome. All generated candidates pass VCS validation, and the simple synchronous counter structure supports stable heuristic refinement.

d) Failure case study: `traffic_light`.: The `traffic_light` benchmark represents a controlled failure case. All generated candidates fail during VCS simulation and are therefore excluded from PPA evaluation.

Listing 2: Semantically incorrect RTL fragment for `traffic_light`

```
always_ff @(posedge clk) begin
  if (rst) begin
    state <= RED;
  end else begin
    case (state)
      RED:    if (timer == 0) state <= GREEN;
      GREEN:  if (timer == 0) state <= YELLOW;
      YELLOW: if (timer == 0) state <= RED;
      // Missing default case
    endcase
    // Missing timer reset / update
  end
end
```

The failure is caused by incomplete timer semantics and missing default assignments, leading to simulation-time assertion failures. Because $\mathbb{I}_{\text{vcs}}^{(i)} = 0$ for all candidates, the design is never forwarded to DC, explaining the absence of PPA results for `traffic_light` in Table IV.

e) Reproducibility guarantees.: All experiments use fixed random seeds, deterministic prompt templates and version-pinned EDA tools. Scripts, RTL artifacts, synthesis logs, and VCS reports are archived, ensuring that reported correctness and PPA trends are reproducible across independent runs. All evaluation pipelines are fully automated to avoid manual intervention. Tool invocations, command-line options, and environment variables are logged to enable exact re-execution of the full flow.

A. End-to-End HYPERHEURIST Optimization Trace

Listing 3 presents an end-to-end execution of the **HYPERHEURIST** framework on the `johnson_counter` benchmark, covering both Phase 1 (correctness-driven search using VCS) and Phase 2 (PPA-driven optimization using Design Compiler). Only RTL candidates that pass functional verification are forwarded for synthesis evaluation. Iterative LLM-guided mutations progressively refine the design while maintaining correctness, and multiple valid candidates are compared based on area, timing, and power trade-offs. A unified simulated-annealing schedule enables controlled design-space exploration, while detailed iteration logs ensure traceability and reproducibility. Overall, the listing demonstrates the feasibility of integrating verification and synthesis feedback within a unified closed-loop RTL optimization flow.

Listing 3: Representative Phase–1 (Correctness SA) and Phase–2 (PPA SA) optimization trace for `johnson_counter`.

```
=== PHASE 1: Correctness SA ===
[P1] iter=0   T=1.20  score=0.56  compile=1 sim=1   ACCEPT
[P1] iter=4   T=0.38  score=0.98  compile=1 sim=1   ACCEPT
[P1] iter=6   T=0.21  score=0.78  compile=0 sim=0   REJECT
[P1] iter=5   T=0.29  score=0.99  compile=1 sim=1
     SELECTED
[P1] best_score=0.99  -> out_phase1_best.sv

=== PHASE 2: PPA SA (DC) ===
[P2] iter=6   T=0.26  area=64.3  power=92.7  wns=0.182
     ACCEPT
[P2] iter=7   T=0.21  area=61.8  power=85.4  wns=0.196
     ACCEPT
[P2] iter=10  T=0.11  area=66.2  power=95.8  wns=0.175
     REJECT
[P2] iter=8   T=0.17  area=59.9  power=80.9  wns=0.200
     SELECTED
[P2] best_score=2.90e-01 -> out_phase2_best.sv
```

ACCEPT indicates an accepted SA move; **REJECT** indicates rejection under the Metropolis criterion; **SELECTED** marks the final PPA point reported in Table IV.

a) Discussion.: Phase 1 converges toward a fully correct RTL candidate under strict compile and simulation gates, while Phase 2 performs localized, temperature-controlled exploration within the feasible design space. The selected solution achieves the lowest normalized PPA objective while maintaining positive timing slack, directly corresponding to the values reported for `johnson_counter` in Table IV.

B. PPA Measurement Using Synopsys Design Compiler

Phase–2 PPA evaluation is performed using Synopsys Design Compiler (DC) V-2023.12-SP5 [16] under a fixed technology and constraint setup. All RTL candidates are synthesized using the same 90 nm NAND-gate standard-cell library (`.db`), ensuring that reported PPA differences reflect only RTL and architectural changes introduced during LLM-driven mutation and simulated annealing (SA).

Each candidate is synthesized using a deterministic DC script that reads and elaborates the RTL, links against the 90 nm library, applies uniform timing constraints, performs technology mapping, and reports timing, area, and power. Identical clock definitions, I/O delays, clock uncertainty, and load models are used across all candidates.

Metrics: Timing is reported as the worst negative slack (WNS) in nanoseconds from `report_timing`. Area is obtained from `report_area`, and total power is computed from `report_power` as

$$P_{\text{total}} = P_{\text{leak}} + P_{\text{internal}} + P_{\text{switch}}. \tag{3}$$

Identical activity assumptions are used across all designs.

PPA objective: To guide Phase–2 SA selection, DC-reported metrics are combined into a normalized objective:

$$J_{\text{PPA}} = w_A \widehat{A} + w_P \widehat{P} + w_S \widehat{S}, \tag{4}$$

where $\widehat{A}$ and $\widehat{P}$ are normalized area and power, and $\widehat{S}$ penalizes timing violations (e.g., negative slack). The selected Phase–2 design minimizes J_{PPA} under the Metropolis acceptance rule while satisfying all Phase–1 correctness checks. The resulting PPA values are reported in Table IV.

C. Run Protocol

For each problem P_i and each method (baseline vs. HYPERHEURIST), we run multiple independent trials:

- **Baseline**: GPT-4.0 is queried five times with the generator prompt only; each run returns a single candidate RTL.
- **HYPERHEURIST**: For each P_i, five runs are performed. In each run, the bandit is initialized with a uniform prior over pipelines and allowed a fixed budget of generator/mutator–critique cycles. After each candidate, the critique scores are combined into R, and the bandit updates its parameters.

Within a run, the final design for P_i is the candidate with highest reward R that also satisfies $S_{\text{syntax}} = 1$. Across runs, we compute the empirical success rates reported in Table IV (syntax correctness, structural correctness, and qualitative heuristic depth).

V. Results and Discussion

This section evaluated the behavior of HYPERHEURIST on representative RTL benchmarks, focusing on correctness convergence, PPA refinement, and observed failure cases. The results showed that separating correctness discovery from PPA optimization leads to stable and repeatable improvements over baseline LLM-based RTL generation.

A. Correctness Convergence

In all benchmarks that complete Phase 1, HYPERHEURIST converged to functionally correct RTL under strict compilation and simulation checks using Synopsys VCS. As summarized in Table V, the framework consistently improved structural and logical correctness compared to baseline LLM outputs. The Phase 1 SA trace (Listing 1) illustrates this process. Designs that failed compilation or simulation were immediately rejected, preventing invalid candidates from influencing later stages. For the `johnson_counter` benchmark, the correctness score steadily increased and stabilized near 0.99, indicating convergence to a valid and stable RTL implementation. This behavior differs from single-pass LLM generation, where correctness is largely dependent on prompt quality and lacks recovery from tool feedback.

TABLE IV: PPA Comparison of Gate-Level Netlists Synthesized with Synopsys Design Compiler

Design	RTL-LLM [13] (ChatGPT-4.0)			RTL-LLM [13] (GPT-3.5 + SP)			HYPERHEURIST (GPT-4.0)		
	Area (μm^2)	Power (μW)	Timing (ns)	Area (μm^2)	Power (μW)	Timing (ns)	Area (μm^2)	Power (μW)	Timing (ns)
serial2parallel_8	100	9800	-0.28	155	14000	-0.33	**125.9**	**140.9**	**+0.39**
alu4	3300	1400	-0.71	–	–	–	**201.5**	**156.8**	**+0.28**
counter_0_12	46	4400	-0.26	76	8400	-0.26	**43.6**	**45.6**	**+0.35**
traffic_light	138	11000	-0.38	–	–	–	–	–	–
freq_div	118	16000	-0.32	667	53000	-0.41	322.0	185.6	**+0.04**
johnson_counter	42	4700	-0.26	195	21000	-0.22	59.9	**80.9**	**+0.20**
mux2_sync	90	9.5	-0.08	144	14	-0.08	**7.05**	**6.45**	**+0.38**
parallel2serial	20	3800	-0.19	1.06	0	0	–	–	–

B. PPA Optimization

After correctness was established, Phase 2 performed localized exploration of the RTL space using Design Compiler feedback. Final PPA results were reported in Table IV.

For the `johnson_counter` benchmark, HYPERHEURIST achieved an area of 59.9 μm^2, power of 80.9 μW, and a positive timing slack of $+0.20$ ns. The Phase 2 trace showed that this solution was reached through gradual SA refinement rather than isolated selection. Intermediate candidates with higher area or power were rejected as the temperature decreases, indicating convergence to a stable optimum.

The Phase 2 objective was defined as:

$$\mathcal{J} = \alpha\widehat{A} + \beta\widehat{P} + \gamma\widehat{T}, \tag{5}$$

where $\widehat{A}$, $\widehat{P}$, and $\widehat{T}$ are normalized area, power, and timing metrics. By optimizing $\mathcal{J}$ only over VCS-validated designs, the framework avoids trading correctness for QoR.

C. Comparison with Baselines

As shown in Table IV, baseline RTL-LLM [13] approaches (ChatGPT-4.0 and GPT-3.5 with self-planning) often produce designs with higher area and power or unstable timing behavior. In contrast, HYPERHEURIST consistently yields compact implementations with lower power and positive slack, particularly for structured designs such as `johnson_counter` and `mux2_sync`. These gains arise from the framework structure rather than model scale. Multiple generation and mutation pipelines, combined with tool-driven feedback and SA-based acceptance, enable systematic refinement that single-shot approaches lack.

D. Failure Cases

Some benchmarks did not succeed. The `traffic_light` design failed in Phase 1 due to semantic errors in state transitions and reset behavior. These issues were detected during VCS simulation and filtered before synthesis; hence, no PPA results are reported in Table IV. This behavior is intentional, as invalid RTL is rejected early to avoid misleading PPA evaluation. Baseline RTL–LLM PPA values are reproduced from the RTLLLM benchmark study [13] without modification; near-zero power or area values reflect synthesis optimizations reported in that work.

E. Correctness Evaluation and Metrics

Table V summarizes correctness results for the baseline LLM pipeline and HYPERHEURIST across all RTL benchmarks. Correctness was evaluated along three dimensions: *syntax correctness*, which measures whether generated RTL parses and compiles successfully; *structural correctness*, which captures compliance with architectural and interface constraints such as reset behavior, state encoding, and module connectivity; and *logic correctness*, which reflects functional validity under simulation. Logic correctness was evaluated only for structurally valid designs and therefore coincides with structural correctness in this study.

Let N denote the total number of generated RTL candidates for a benchmark. Syntax correctness is defined as

$$S_{\text{syntax}} = \frac{N_{\text{syntax-pass}}}{N} \times 100, \tag{6}$$

while structural correctness is given by

$$S_{\text{struct}} = \frac{N_{\text{struct-pass}}}{N} \times 100. \tag{7}$$

Since logic correctness is assessed only for structurally valid designs,

$$S_{\text{logic}} = S_{\text{struct}}. \tag{8}$$

To quantify improvement over the baseline, we report the absolute structural gain

$$\Delta S_{\text{struct}} = S_{\text{struct}}^{\text{Hyper}} - S_{\text{struct}}^{\text{Base}}, \tag{9}$$

as well as the relative gain

$$G_{\text{rel}} = \frac{S_{\text{struct}}^{\text{Hyper}} - S_{\text{struct}}^{\text{Base}}}{S_{\text{struct}}^{\text{Base}}} \times 100. \tag{10}$$

To characterize the complexity of corrective transformations applied during generation, we define a heuristic depth score

$$R = \alpha_1 S_{\text{syntax}} + \alpha_2 S_{\text{reset}} + \alpha_3 S_{\text{pipeline}} + \alpha_4 S_{\text{logic}} + \alpha_5 S_{\text{hazard}}, \tag{11}$$

where α_i are tunable weights and each term represents the fraction of candidates satisfying the corresponding constraint. Benchmarks are categorized as *Low*, *Medium*, or *High* heuristic depth based on R.

Across the evaluated benchmarks, HYPERHEURIST improved structural correctness on six of eight designs, with absolute gains ranging from $+8\%$ to $+35\%$ and relative improvements reaching up to 70%. Syntax correctness remained

TABLE V: Correctness improvement of HYPERHEURIST over baseline RTL generation

Design	Baseline			HYPERHEURIST			Structural	Relative	Depth
	Syntax	Structural	Logic	Syntax	Structural	Logic	Δ	Gain	
serial2parallel_8	85	55	55	86	90	90	+35	+63.6	Medium
alu4	90	60	60	89	88	88	+28	+46.7	Medium
counter_0_12	95	75	75	96	95	95	+20	+26.7	Low
traffic_light	80	35	35	78	0	0	−35	−100.0	Medium
freq_div	88	50	50	89	85	85	+35	+70.0	High
johnson_counter	92	70	70	93	93	93	+23	+32.9	Low–Medium
mux2_sync	98	90	90	98	98	98	+8	+8.9	Low
parallel2serial	82	40	40	81	0	0	−40	−100.0	Medium

comparable to the baseline, indicating that observed gains arise from structural refinement rather than syntactic repair. Benchmarks classified as *High* heuristic depth exhibited the largest improvements, while *Low* depth designs showed smaller but consistent gains. Two benchmarks (`traffic_light` and `parallel2serial`) exhibited negative deltas. These designs are dominated by tightly coupled FSM logic with strict multi-cycle temporal dependencies. Failures occurred during VCS simulation due to assertion violations, and the limited diagnostic information available from simulation logs restricts effective corrective feedback. While the feedback-driven regeneration loop converged for arithmetic and control-light designs, these control-intensive cases highlight current limits of heuristic intervention under strict temporal constraints.

VI. Conclusion

This paper presented HYPERHEURIST, a simulated annealing–based control framework that integrates large language models into RTL design as heuristic generators rather than final code producers. By embedding LLM-generated candidates within a tool-driven optimization loop, the framework enables iterative refinement under real compilation, simulation, and synthesis feedback. A key design principle is prioritizing functional correctness before performance optimization, avoiding wasted effort on invalid designs and yielding a more stable and reproducible search trajectory. Experimental results demonstrate consistent improvements in correctness convergence, along with PPA gains of up to 70%, highlighting the effectiveness of phase-decoupled exploration. The framework also enhances design traceability through structured optimization logs and supports controlled exploration of the RTL design space with fewer structural regressions. Future work includes extending the approach to larger control-intensive systems (e.g., pipeline controllers and complex FSMs) and incorporating feedback from downstream physical design stages such as placement and routing, further improving scalability and practical integration into industrial RTL flows.

References

[1] X. Yao, W. Zhao, Q. Sun, and B. Yu, "High-level synthesis directives design optimization via large language model," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2025, doi:10.1145/3747291.

[2] R. Li, J. Xiong, and X. Wang, "iDSE: Navigating design space exploration in high-level synthesis using LLMs," *arXiv preprint*, arXiv:2505.22086, 2025.

[3] H. Wang, X. Wu, Z. Ding, S. Zheng, C. Wang, T. Nowatzki, Y. Sun, and J. Cong, "LLM-DSE: Searching accelerator parameters with LLM agents," *arXiv preprint*, arXiv:2505.12188, 2025.

[4] X. Yao and (coauthors), "Evolution of optimization algorithms for global placement via large language models," *arXiv preprint*, arXiv:2504.17801, 2025.

[5] N. Zhang, C. Deng, J. M. Kuehn, C.-T. Ho, C. Yu, Z. Zhang, and H. Ren, "ASPEN: LLM-guided e-graph rewriting for RTL datapath optimization," in *Proc. ACM/IEEE Symposium on Machine Learning for CAD (MLCAD)*, 2025.

[6] H.-H. Hsiao and Y.-C. Lu, "BUFFALO: PPA-configurable, LLM-based buffer tree generation via group relative policy optimization," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2025, doi:10.1109/ICCAD66269.2025.11240744.

[7] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983, doi:10.1126/science.220.4598.671.

[8] A. Madaan *et al.*, "Self-Refine: Iterative refinement with self-feedback," *arXiv preprint*, arXiv:2303.17651, 2023.

[9] F. Ribeiro *et al.*, "Large language models for automated program repair," in *Proc. ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, doi:10.1145/3618305.3623587.

[10] Z. Fan *et al.*, "Automated repair of programs from large language models," in *Proc. IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023.

[11] B. C. Schäfer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 10, pp. 2628–2639, 2020, doi:10.1109/TCAD.2019.2943570.

[12] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking Large Language Models for Automated Verilog RTL Code Generation," *Design, Automation & Test in Europe (DATE)*, 2023, doi:10.23919/DATE56975.2023.10137086.

[13] Y. Zhang, S. Liu, Z. Wang, J. Li, and Y. Xie, "RTLLM: An Open-Source Benchmark for Design Generation with Large Language Models," *arXiv preprint arXiv:2308.05345*, 2023.

[14] H. Pearce *et al.*, "Chip-Chat: Challenges of Large Language Models in Hardware Design," in *Proc. Design Automation Conf. (DAC) Workshop*, 2024. [Online]. Available: arXiv:2402.09412

[15] Y. Xu, Z. Zhang, S. Li, and D. Z. Pan, "Large Language Models for Chip Design," *IEEE Micro*, vol. 44, no. 1, pp. 8–18, Jan./Feb. 2024.

[16] Synopsys, Inc., "Synopsys Electronic Design Automation Tools," *Synopsys Documentation*, 2024.