

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

LARGE LANGUAGE MODELS IN APPROXIMATE CIRCUIT DESIGN

VELKÉ JAZYKOVÉ MODELY V NÁVRHU APROXIMATIVNÍCH OBVODŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MARTIN TOMAŠOVIČ**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2026

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

170086

Institut:	Department of Computer Systems (DCSY)
Student:	**Tomašovič Martin, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Large Language Models in Approximate Circuit Design**
Category:	Artificial Intelligence
Academic year:	2025/26

Assignment:

1. Familiarize yourself with available large language models (LLMs) and their applications in digital circuit design. Focus on the design of approximate arithmetic circuits.
2. Propose a method that uses LLM in the design or optimization of approximate circuits.
3. Implement the proposed method using available LLMs and other software libraries.
4. Experimentally verify the proposed method.
5. Evaluate the achieved results and discuss possible directions for continuing the project.

Literature:

- According to the instructions of the supervisor.

Requirements for the semestral defence:

- Items 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Sekanina Lukáš, prof. Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2025
Submission deadline:	20.5.2026
Approval date:	31.10.2025

Abstract

Approximate circuits are an effective alternative to accurate circuits. They offer lower energy consumption, lower latency, and lower area at the cost of controlled error. In this work, we specifically focus on the design of 8-bit approximate multipliers using large language models (LLMs), which we combine with evolutionary algorithms. The work researches the use of LLMs for circuit generation across different settings. We selected a model, fine-tuned it, and studied its properties. Then we proposed algorithms that call an LLM in a cycle or as part of a multiobjective evolutionary algorithm, which improve not only the success rate of generating valid circuits but also their quality. We propose a coevolution approach. It evolves a population of circuits and a population of prompt templates that control LLM. Conducted experiments show that the proposed method can find circuit designs with a better trade-off between error and area estimates than existing circuits in the EvoApproxLib library.

Abstrakt

Aproximatívne obvody predstavujú alternatívu k presným digitálnym obvodom, pričom za cenu kontrolovanej chyby dosahujú nižšiu spotrebu, latenciu a plochu. V tejto práci sa zameriavame na návrh aproximatívnych 8-bitových násobičiek pomocou veľkých jazykových modelov (LLM), ktoré kombinujeme s evolučnými algoritmami. V práci skúmame využitie LLM pre generovanie obvodov s rôznymi nastaveniami. Vybraný model sme dotrénovali a skúmali jeho vlastnosti. Následne sme navrhli algoritmy, ktoré volajú LLM v cykle, či ako súčasť multikriteriálneho evolučného algoritmu, čo zlepšuje nielen úspešnosť generovania platných obvodov, ale aj ich kvalitu. Navrhujeme evolučný prístup, v ktorom sa vyvíja populácia obvodov a populácia šablón promptov riadiacich LLM. LLM je využitý ako operátor mutácie, ktorý generuje nové návrhy obvodov. Experimenty ukazujú, že navrhovaná metóda dokáže nájsť riešenia s lepším kompromisom medzi chybou a odhadom plochy v porovnaní s existujúcimi obvodmi z knižnice EvoApproxLib.

Keywords

large language models, approximate circuit, circuit design, few-shot prompting, fine-tuning, evolution

Kľúčové slová

veľké jazykové modely, aproximatívne obvody, návrh obvodov, dotazovanie s niekoľkými príkladmi, dotrénovanie, evolúcia

Reference

TOMAŠOVIČ, Martin. *Large Language Models in Approximate Circuit Design.* Brno, 2026. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Rozšírený abstrakt

Táto práca skúma využitie veľkých jazykových modelov (LLM) pri návrhu aproximatívnych obvodov. Konkrétne sme sa zamerali na 8-bitové násobičky bez znamienka. Aproximatívne násobičky sa dajú využiť napríklad v akcelerátoroch neurónových sietí. Jazykové modely skúmame samostatne a následne ich kombinujeme s evolučnými algoritmami, kde LLM funguje ako operátor mutácie a rekombinácie. Základná myšlienka je, že LLM využíva znalosti získané predtrénovaním na navrhovanie inteligentných úprav obvodov.

V tejto oblasti existuje niekoľko prác, ktoré používajú podobnú metodológiu. GPTAC [34] je najbližšia práca, kde dotrénovaný model GPT-2 generuje aproximatívne obvody. GPTAC využíva algoritmy, ktoré upravujú výstup počas inferencie s cieľom ho zlepšiť a zvýšiť generalizáciu. V porovnaní s GPTAC sa náš prístup odlišuje využitím formátu logických hradiel namiesto formátu štandardných buniek. Okrem odlišného formátu reprezentácie obvodov sme sa zamerali na odlišné modely využívajúce znalosti z predtrénovania.

Pred návrhom finálnej metódy sme realizovali sériu experimentov s cieľom preskúmať vlastnosti LLM pred a po dotrénovaní menšieho 7B modelu v spomínanej doméne. Obvody sú reprezentované pomocou matematických rovníc na úrovni logických hradiel. Hradlá AND, OR, XOR a NOT sú reprezentované notáciou prevzatou z programovacieho jazyka C. Tento formát je pre človeka čitateľný a pochopiteľný. Na základe prvotných experimentov sme zistili, že podobnosť s programovacími jazykmi dovoluje LLM využiť znalosti z predtrénovania. Model, ktorý dosahoval najlepšie výsledky pri generovaní návrhov obvodov bol predtrénovaný pre riešenie matematických úloh a generovanie kódu.

Následne sme skúmali vplyv few-shot promptovania a počtu príkladov obvodov v jednom prompte na kvalitu generovaných obvodov. Experimenty ukázali, že aj bez dodatočného trénovania dokáže LLM generovať nové validné návrhy, avšak s vysokým počtom syntaktických a sémantických chýb. Preto sme vybraný 7B model dotrénovali na doménovo špecifických dátach. Tento model sme testovali s rôznymi nastaveniami teploty. Tiež sme navrhli vlastnú transformáciu na úrovni vrstvy neurónovej siete. Pri aplikácii na správnu vrstvu táto transformácia mení distribúciu generovaných návrhov a umožňuje modelu vytvárať obvody, ktoré by za štandardných podmienok nevygeneroval.

Model sme dotrénovali na vlastnom datasete pripravenom z obvodov EvoApproxLib. Následne bol model testovaný v evolučnom algoritme a v iteratívnej slučke s dvomi rôznymi stratégiami. Tieto experimenty ukázali schopnosť modelu generovať obvody s nízkou chybou alebo malou plochou, avšak pri hľadaní vyvážených kompromisov medzi chybou a plochou odhalili jeho obmedzenia.

Na základe získaných poznatkov sme navrhli finálnu koevolučnú metódu využívajúcu väčšie modely bez ďalšieho dotrénovania. LLM v nej funguje ako operátor mutácie a rekombinácie. Koevolučná metóda pomocou mutácie vytvára nové návrhy obvodov z aktuálnej populácie obvodov v každej generácii. Obvody sa generujú na základe šablón promptov, ktoré sa vyvíjajú spolu s obvodmi. Každá šablóna má vlastnú fitness, ktorá závisí na kvalite obvodov ktoré vygenerovala. Meranie chyby obvodov je vykonané konverziou do programovacieho jazyka C. Plocha sa odhaduje na základe počtu jednotlivých typov hradiel. Navrhovaná metóda dokáže nájsť riešenia s porovnateľným a aj lepším kompromisom medzi chybou a odhadom plochy v porovnaní s existujúcou EvoApproxLib knižnicou. Experimenty identifikovali 25 návrhov rozširujúcich globálnu Pareto frontu pre metriku WCE a 23 návrhov pre metriku MSE.

Hlavným prínosom práce je systematické preskúmanie možností využitia LLM pre automatický návrh aproximatívnych obvodov a narhnutý koevolučný algoritmus. Práca zároveň

prináša poznatky o vplyve reprezentácie obvodov, few-shot promptovania, dotrénovania modelov a transformácií neurónových vrstiev na kvalitu generovaných obvodov.

Large Language Models in Approximate Circuit Design

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of prof. Ing. Lukáš Sekanina, Ph.D.

.....
Martin Tomašovič
May 20, 2026

Acknowledgements

I would like to sincerely thank my supervisor, prof. Ing. Lukáš Sekanina, Ph.D., for his mentoring, support, and valuable feedback throughout this work.

This work was supported by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254) and Czech Science Foundation project 24-10990S.

Contents

1 Introduction **6**

2 Background **7**
2.1 Approximate arithmetic circuits 7
2.2 Large language models 10

3 Related work on approximate circuit design using AI **13**
3.1 Approximate circuit generation using generative pre-trained model 13
3.2 PrefixLLM 16
3.3 Cartesian genetic programming 18

4 Proposed circuit approximation method based on LLM **21**
4.1 Circuit representation 21
4.2 Datasets 22
4.3 Large language models 22
4.4 Measurements 23
4.5 Few-shot prompting 23
4.6 Training 24
4.7 Layer inversion 25
4.8 Refinement loop 26
4.9 Evolutionary algorithms 26
4.10 Coevolution 32
4.11 Circuit types 35

5 Implementation **36**
5.1 Format conversion 36
5.2 Circuit error and area computation 36
5.3 Dataset preparation 37
5.4 Fine-tuning 38
5.5 Algorithms calling LLM 39

6 Experiments **41**
6.1 Model comparison 41
6.2 Impact of few-shot example count on circuit design 44
6.3 Fine-tuned model experiments 45
6.4 Bit-width generalization of circuit generation 46
6.5 LLM in a loop 48

6.6 Evolution with fine-tuned model experiments 51
6.7 Evolution with API models experiments 53
6.8 Experiments summary 60

7 Conclusion **64**

Bibliography **66**

A Experiment results **70**
A.1 Few-shot model comparison 70
A.2 Fine-tuned model experiments results 70

List of Figures

2.1 Transformer schema. On the left side, there are encoder blocks, and on the right side, there are decoder blocks. Transformer inputs are positionally encoded. The positionally encoded inputs are then processed by stacks of N encoder blocks and N decoder blocks. Each block consists of multiple layers of self-attention and point-wise, fully connected layers. Taken from [32]. . . 11

3.1 GPTAC tokenization process (Top-left: Character-level vectorization; Bottom-left/right: Node-level vectorization for logic-gates/standard-cell-netlist). Taken from [34]. 15
3.2 PrefixLLM Iterative framework for prefix circuit synthesis. Taken from [33]. 17
3.3 PrefixLLM: An iterative design space exploration framework for prefix circuit optimization. Taken from [33]. 18
3.4 CGP representation of a two-bit multiplier. Taken from [23]. 19
3.5 Schema of approximation using CGP. Taken from [23]. 19

4.1 Few-shot pipeline with format processing. 24
4.2 A diagram of an algorithm for automatic loop refinement of generated circuits. 29
4.3 A high-level diagram of the prompt and circuit coevolution algorithm. . . . 30
4.4 A diagram of the prompt crossover and mutation algorithm as part of co-evolution through a language model. 30
4.5 A diagram of the circuit generation algorithm as part of the coevolution through the language model. 31
4.6 The proposed circuit approximation method that utilizes a coevolutionary algorithm evolving a population of circuits (left-hand side) and a population of circuit- producing templates (right-hand side). The circuit- producing templates population provides templates for generating new circuits. After the circuit fitness measurement, template fitness functions are measured and used by the template population. 33

6.1 Results of few-shot prompting. The prompt contains two example circuits; the target is to minimize MSE while preserving area (cost). Columns express ratios between the minimal MSE from the given two example circuits and the MSE of the generated circuit, similarly for cost. Part (a) shows the results of prompting with the prompt template version 1, and Part (b) shows version 2. 43
6.2 Measured differences between a) cost and b) MSE of generated circuit designs and the minimal, average, and maximal original circuit parameters. 45

6.3 Pareto dominant circuits. Comparing circuits generated by LLM with different settings to the ground truth from the EvoApproxLib dataset. On the top (a), there are separated Pareto fronts for each type of setting, and on the bottom (b), there is a global Pareto front. 47
6.4 Syntactical and semantical correctness of the circuit designs generated with LLM with various settings of temperature and layer inversion. 48
6.5 Progress of Qwen2.5-Coder-7B-Instruct fine-tuning for 8 epochs. 49
6.6 Comparison of 500 iterations with fine-tuned LLM. 50
6.7 Comparison of 500 and 1000 iterations. Refinement loop algorithms with fine-tuned LLM. 51
6.8 Layer inversion at layer 8. Refinement loop algorithms with fine-tuned LLM. 52
6.9 Layer inversion at layer 24. Refinement loop algorithms with fine-tuned LLM. 52
6.10 The *WCE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed Evolution with fine-tuned model (brown squares) or taken from EvoApproxLib (blue dots or light blue crosses). 53
6.11 The *MSE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed Evolution with fine-tuned model (brown squares) or taken from EvoApproxLib (blue dots or light blue crosses). 54
6.12 CoEA with different initialization. 55
6.13 Final Pareto fronts obtained from 10 runs of HC, CoEA-3-5, and CoEA-1-15. The same initial population of circuits was used in the i-th run of CoEA algorithms. The target specification is depicted with a red cross. 57
6.14 The hypervolume indicator computed from the final Pareto front constructed from 10 independent runs of HC, CoEA-3-5, and CoEA-1-15. 57
6.15 The progress of evolution of CoEA-3-5 and CoEA-1-15, where the hypervolume indicator is represented as a box plot of the best solutions (in 10 runs) reached in a given generation. 58
6.16 The *WCE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed CoEA or taken from EvoApproxLib. The united global Pareto front contains non-dominated solutions generated by CoEA (green squares) and taken from EvoApproxLib (blue dots). The remaining points are dominated. The target specifications provided to CoEA are depicted with red crosses. . 59
6.17 The *MSE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed CoEA or taken from EvoApproxLib. The united global Pareto front contains non-dominated solutions generated by CoEA (green squares) and taken from EvoApproxLib (blue dots). The remaining points are dominated. The target specifications provided to CoEA are depicted with red crosses. . 59
6.18 Synthesis of circuits produced by CoEA for WCE. The proposed designs are compared to reference datasets. 62
6.19 Synthesis of circuits produced by CoEA for MSE. The proposed designs are compared to reference datasets. 63

A.1 Results of few-shot prompting. The prompt contains two example circuits; the target is to minimize MSE while preserving area (cost). The data in these graphs are filtered to fit the condition that the MSE of the generated circuit is lower or maximally 10% higher than the minimal MSE of the circuit from the prompt, and the same condition for cost. Columns express ratios between the maximal MSE from the given two example circuits and the MSE of the generated circuit, similarly for cost. Part a) shows the results of prompting with the prompt template version 1, and Part b) shows version 2. 71
A.2 A scatter plot showing MSE to cost comparison. The data in these graphs are filtered to fit the condition that the MSE of the generated circuit is lower or maximally 10% higher than the minimal MSE of the circuit from the prompt, and the same condition for cost. Part a) shows the results of prompt template version 1, and Part b) shows version 2. 72
A.3 Comparison of Pareto fronts for each setting individually. 73

Chapter 1

Introduction

Approximate circuits are digital circuits designed to operate inexactly [14, 26]. They trade controlled error for reduced latency, lower power consumption, or smaller chip area, rather than accurate circuits. These advantages are utilized in error-tolerant applications such as sensor data, image, audio processing [24, 23], or deep neural networks, where approximate implementations can reduce power consumption and latency in exchange for an acceptable error of the output [1]. Specifically, approximate multipliers and adders are the most crucial circuits for the aforementioned applications.

The approximate circuits can be designed by conventional approaches [13, 14], but evolutionary design methods have shown very strong performance in this task [21, 4, 22]. One of the recent approaches to automated approximate circuit design uses large language models (LLMs). This method uses a domain-specific generative pre-trained model for approximate circuit design (GPTAC) [34]. Nevertheless, the capabilities of LLMs for approximate circuit design remain largely unexplored, particularly when combined with evolutionary techniques.

This thesis aims to investigate the use of large language models for the design of approximate arithmetic circuits. First, the relevant literature of existing solutions was studied, and then new solutions were proposed and tested. To investigate their abilities, the best pretrained models of the same size but different types were selected. Their capability to generate new circuits was tested by few-shot prompting. In addition, the most suitable model for fine-tuning was identified. The fine-tuned model was tested for zero-shot prompting with different settings. Further experiments were proposed, in which the fine-tuned model is a central component of iterative and evolutionary frameworks. The final experiments employed significantly larger pre-trained models in a coevolutionary algorithm.

Chapter 2 provides background on concepts related to this thesis, including approximate circuits and large language models. Chapter 3 summarizes existing state-of-the-art solutions. Chapter 4 proposes methods for circuit design using LLM; the implementation details are described in Chapter 5. Chapter 6 describes conducted experiments and their results. We conclude with Chapter 7, where we recapitulate the results and the most interesting findings.

Chapter 2

Background

In the beginning, we introduce the background and define the key concepts. The most important concepts in this work are approximate circuits and large language models (LLMs).

Approximate circuit definition, types, and methods for approximate circuit design and measurement are all described in this chapter. The second part focuses on the LLMs and the transformer architecture on which they are built. We will mention different categories of LLMs. After that, we perform a short deep dive into the key concept of self-attention, the most crucial idea behind LLMs.

2.1 Approximate arithmetic circuits

An approximate arithmetic circuit is a circuit, or, in other words, a set of connected logic gates in the form of an acyclic directed graph, which performs an arithmetic operation such as multiplication, addition, subtraction, or others. In approximate arithmetic circuits, approximation denotes trade-offs among circuit parameters such as area, delay, energy consumption, and error. These trade-offs are convenient for different use cases, such as reducing power consumption in signal processing applications or addressing the current high demand for deep learning models. We focused on combinational gate-level approximate circuits [24], which constitute the largest set of approximate circuits; however, iterative or sequential ones have also been explored in the study [5]. A combinational circuit is a circuit whose output depends only on its current inputs.

2.1.1 Circuit representation

In general, circuit approximation can be performed at one of several abstraction levels, such as the transistor level, gate level, register-transfer (RT) level, behavioral level, or look-up table (LUT) level. A circuit can be represented in many formats at different abstraction levels. Each format has its own advantages and disadvantages. Circuit formats offer different perspectives on the same concept. Examples of formats are: a diagram, HDL (Verilog) code, programming (C) code, netlist, mathematical equations, and others.

Diagrams are human-readable, while it is more difficult to serve them to a language model as input. To measure circuit properties, the diagram needs to be converted into another format.

HDL formats are a standard way to design circuits; many tools exist to convert them to a specific technology and simulate the circuits. They are also well human-readable. From the perspective of being input to an LLM, there is one minor disadvantage: syntactic

overhead compared to other formats. One must also consider that, currently, most of the available pretrained LLMs do not perform well at generating HDL code due to the scarcity of HDL code in LLM training datasets [29].

A netlist describes circuit components (e.g., a resistor and its resistance) and the connections between them. They often represent the circuits in certain technologies. The synthesis of an HDL-represented circuit generates the HDL representation. This representation is still human-readable but less clear than HDL.

Programming language representations are used for error analysis because the circuit can be executed as standard code. For each input combination, we obtain the same output as we would from the real circuit. These outputs are then compared to the correct outputs, the ones of an exact circuit.

Mathematical equations are human-readable, but they are not the standard for representing circuits. They need to be converted to other formats for use with standard electronic design tools.

2.1.2 Design approaches

Now that we have clarified what an approximate circuit is, we need to know how to make them. The approximation methods can be divided into three groups: manual design by modifying exact circuits, developing new approximation schemes for specific applications, and automated methods using design space exploration algorithms [24].

Manual design is very time-consuming, requires a professional designer, and is only suitable for small circuits. These smaller approximated circuits can then be used as part of larger circuits.

The second approach starts from a greenfield with no example and is based on a construction scheme for a specific class of problems. New circuits are designed based on mathematical principles. This approach designs them from scratch, whereas the next approach uses existing circuits. An example is the replacement of multiplication by weighted addition via linearization of multiplication.

Finally, the last approach, the one that will be the target of this thesis. The use of algorithms for automated circuit design. Here, automated circuit optimization or resynthesis algorithms are used without any need for human designer intervention. Typically, there is a starting circuit that is iteratively modified through pruning, component replacement, or other techniques. In the end, a set of Pareto non-dominated circuits is obtained. This approach also brings some challenges. One of them is efficient circuit evaluation.

2.1.3 Approximation techniques

Here, we describe techniques for obtaining an approximate circuit. The techniques transform existing circuits into approximate ones with the required properties.

One technique is the voltage overscaling (VOS) [19]. First, without changing the circuit structure, the supply voltage is reduced. This results in power consumption reduction. It causes problems such as increased delay, and delays on the critical path can cause timing errors, resulting in output errors. The error also influences the most significant bits (MSBs). An error at MSB causes large output errors [6].

Other techniques are commonly used, for example, removing, adding, or modifying elements of an accurate circuit. Simplification of Karnaugh maps can also result in an approximate circuit [17].

Multiplication or square root computation is more complex than addition, and thus, there are different approximation approaches. The functions are converted into simpler operations, as in Mitchell's binary logarithm-based algorithm, which uses adders to implement multipliers.

In practice, the approximation techniques are combined to produce an approximate circuit with the required properties [14].

2.1.4 Error analysis

For approximate circuits, the most important metric is the error with respect to the exact implementation. There exist many error metrics, e.g., the mean squared error (MSE) (eq. 2.1), mean absolute error (MAE) (eq. 2.2), worst case error (WCE) (eq. 2.3) and others [24, 34].

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(D_{orig}^{(i)} - D_{approx}^{(i)})^2 \tag{2.1}$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|D_{orig}^{(i)} - D_{approx}^{(i)}| \tag{2.2}$$

$$WCE = \max_{i=1,\ldots,N}|D_{orig}^{(i)} - D_{approx}^{(i)}| \tag{2.3}$$

D denotes a decimal number, where $D^{(i)}$ represents output value of the i-th input combination.

N is 2^{2n} represents all possible combinations of two n-bit inputs.

Most analysis methods use only estimated errors, because computing the exact error becomes more time-consuming with bigger circuits. The estimated error is measured by circuit simulation across a subset of input vectors. This subset should be of appropriate size and reasonably inclusive.

Another way to compare and analyze circuits is to analyze them formally. Formal verification techniques often rely on equivalence checking, i.e., checking whether the circuit's mathematical model meets the model's design specification. The main approaches in formal equivalence checking are satisfiability (SAT) solvers and Ordered Binary Decision Diagrams (ROBDD). In either case, an auxiliary circuit is first constructed and then analyzed. The auxiliary circuit is called a miter.

2.1.5 Multi-objective optimization

When circuits are compared by one error metric, it is trivial to say which circuits are better and which are worse. However, when circuits are evaluated using multiple error metrics, deciding which is better becomes more complex. To solve this problem, the Pareto dominance relation is used. Before further explaining Pareto dominance, consider that we can evaluate and compare circuits not only by their error metrics but also by area, delay, power consumption, etc. So, finding circuits with certain parameters can be viewed as an optimization problem, specifically a multi-objective optimization problem. For example, the objectives can be to simultaneously minimize MSE and power consumption [24].

After measuring circuit metrics, the Pareto front is constructed, with non-dominated circuits lying on it. The formal definition of dominance criterion when we want to minimize

the objectives is formally defined in Equations 2.4 and 2.5. Note that we may want to maximize some objectives; in that case, the direction of inequalities is inverted in the definition for those objectives.

Given x, y and objectives $f_1, \ldots, f_n$ to minimize:
$x \succ y$ (x dominates y) if and only if:
$$\forall i \in \{1, \ldots, n\} : f_i(y) \leq f_i(x) \tag{2.4}$$
$$\exists j \in \{1, \ldots, n\} : f_j(y) < f_j(x) \tag{2.5}$$

2.2 Large language models

The large language model (LLM) is a deep learning neural model trained for natural language processing (NLP) tasks. Usually, by LLM in the literature, it is meant a model with many trainable parameters, trained on a vast amount of data for an NLP task. Language models were built on different concepts (recurrence) before the introduction of the attention mechanism. Current LLMs are neural networks with a Transformer architecture, where the most important part is an attention layer. The transformer architecture was first proposed in the study [32].

LLMs are currently the most popular type of neural network model. The general public often confuses them with general artificial intelligence. Their main advantage over the other approaches is that they model natural language very well due to the aforementioned attention mechanism. The attention blocks capture relationships between different parts of a text. This mechanism resembles the attention we know naturally, when we focus on the more important parts of perception while omitting the less important ones. However, the machine learning attention works differently. It assigns different parts of the text different weights. When larger models with this mechanism are trained, they begin to exhibit generalization abilities, and they can accomplish tasks for which they were not directly trained.

LLMs are now being intensively researched and applied in different areas. Many such applications are used to create software. Although hardware design generation using this technology is currently under research, there is significant potential to investigate it further.

2.2.1 Transformer architecture

The transformer architecture [32] is depicted in Figure 2.1. Parts of this section are partially based on the author's bachelor's thesis [30]. The information flow is as follows. The input sequences are tokenized and converted to embeddings using the embedding layer. After that, the embeddings are positionally encoded. The purpose of the positional encoding is to incorporate information about the absolute position of the tokens in the sequence. To generate the input representation $X = W + P \in \mathbb{R}^{l \times d}$, matrices of word embeddings W and positional encodings P are summed together, l is the length of the sentence, and d is the embedding dimension.

The transformer consists of two main types of blocks: a decoder and an encoder. The input for the transformer is a sequence $x_1, x_2 \ldots x_n$. The encoder transforms this sequence to $z_1, z_2 \ldots z_n$ and the decoder transforms a sequence of zs to a sequence $y_1, y_2 \ldots y_n$, while previously generated output tokens y are fed to the model. It is important to note that these blocks are present multiple times in the architecture as parallel blocks.

Each transformer block is composed of multiple layers. Each layer in the encoder and the decoder contains a fully connected feed-forward network, which is applied separately and identically to each position. For each position, a linear transformation is performed; this transformation is defined in equation 2.6, where FFN is the fully connected feed-forward network, $\mathbf{W}_1, \mathbf{W}_2$ are word embeddings, b_1, b_2 are the bias vectors, and x is the input vector. Transformations differ at the layer level, but are consistent at different positions [32, 15].

$$FFN(\mathbf{x}) = max(0, \mathbf{x}\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2 \tag{2.6}$$

To compare the attention and previously used recurrent mechanisms for language modeling. The key advantage of attention layers is that they reduce the number of computational steps information must pass through from one part of the network to the other. Another important advantage is that attention allows more computation to be parallelized.

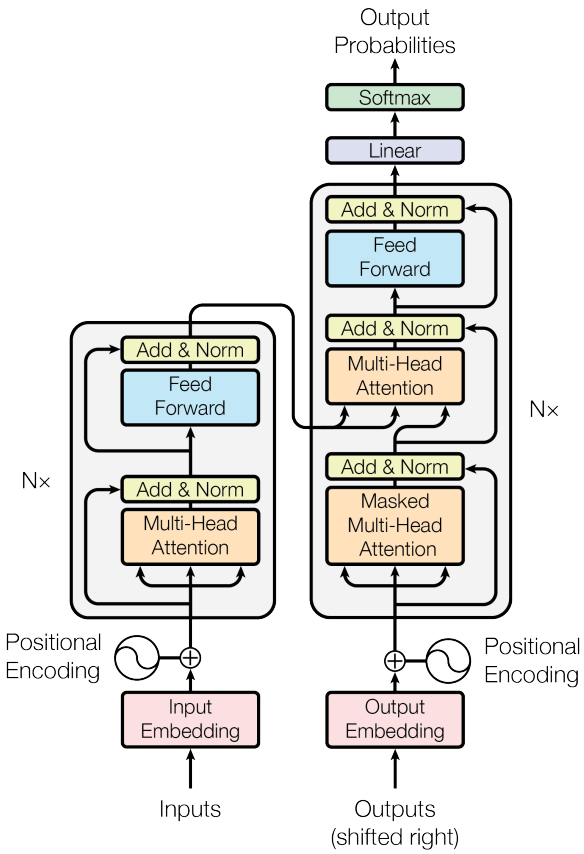


Figure 2.1: Transformer schema. On the left side, there are encoder blocks, and on the right side, there are decoder blocks. Transformer inputs are positionally encoded. The positionally encoded inputs are then processed by stacks of N encoder blocks and N decoder blocks. Each block consists of multiple layers of self-attention and point-wise, fully connected layers. Taken from [32].

2.2.2 Autoregressive models

Large language model architectures can be divided into three types: encoder-decoder, encoder-only, and decoder-only architectures. While the encoder-decoder architecture is

used in the original transformer-based LLMs, which were intended for language translation, the decoder-only architecture is used and researched for language modeling and text generation.

Masked language models such as BERT [9] use an encoder-only architecture. These models are used for tasks that require filling in missing parts of a text. In this architecture, each generated token depends on the entire sequence, on all the other tokens, both previous and future.

Decoder-only LLMs generate text autoregressively. That means they generate one token at a time. Each next token depends only on the previous tokens, this is formally expressed in equation 2.7, where x_i is a token, $p(x_i)$ is the probability of generating the token x_i, $p(x_1, x_2, \ldots, x_n)$ is the probability of a generated token sequence and $p(x_i|x_1, x_2, \ldots, x_{i-1})$ is a conditional probability of generating the token x_i given the sequence of previously generated tokens $x_1, x_2, \ldots, x_{i-1}$.

$$p(x_1, x_2, \ldots, x_n) = p(x_1) \prod_{i=2}^{n} p(x_i|x_1, x_2, \ldots, x_{i-1}). \tag{2.7}$$

2.2.3 Self-attention

Self-attention layer basically converts input vectors $\mathbf{x}_i$ to output vectors $\mathbf{z}_i$. For this conversion, the layer uses an attention matrix generated from the input vector. The attention matrix produces a weighted average of the input feature. The attention matrix is generated by measuring the similarity between two locations in the input sequence using the dot product between the features at those two locations. Then a softmax function is applied to manage the normalization. This approach combines information about the similarity between locations within the sequence with the content of the sequence itself [31].

The self-attention uses trainable matrices to produce query, key, and value vectors. Each input vector $\mathbf{x}_i$ generates three different vectors: query, key and value $(\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i)$. These three vectors are obtained by projecting the input vector, $\mathbf{x}_i$, at time i on the learnable weight matrices $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$. As a result of this projection we get $\mathbf{q}_i$, $\mathbf{k}_i$ and $\mathbf{v}_i$. The trainable parameters of the model are the weights in the weight matrices.

However, in reality, the query, key, and value vectors are computed using matrix operations. The input matrix $\mathbf{X} \in \mathbb{R}^{l \times d}$, where l is the maximum length of the sequence and d is the dimension of the inputs, combines with each of the query, key and value matrices for each token i as a single computation given by equation 2.8. In the equation, d_k is the dimension of the keys, and the scaling factor $\frac{1}{\sqrt{d_k}}$ in the equation prevents very small gradients from softmax when the dot product is large.

$$attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{2.8}$$

Returning to the vector representation, the query, key, and value vectors have the following roles in self-attention. The query vector of token i, $\mathbf{q}_i$, combines with every other key vectors $\sum_{j=0}^{l} \mathbf{q}_i {\mathbf{k}_j}^T$ to influence the weights for its own output, $\mathbf{z}_i$. The role of the value vector of token i, $\mathbf{v}_i$, is to extract information. For this purpose, it is combined with the query-key scores to get the output vector $\mathbf{z}_i$. Finally, the key vector of token i, $\mathbf{k}_i$, is intended to be matched with every other query vector, to get similarity with the query. Further, this influences the output through query-key scoring [15].

Chapter 3

Related work on approximate circuit design using AI

With the problem defined and the theoretical background described, this chapter reviews existing approaches to the automated design of approximate circuits using AI methods.

First, we examine Generative Pre-trained Model for Approximate Circuit Design Exploration, a work that shares the most traits with this thesis. Then we explore another use of LLM in circuit exploration, but this time it is not about approximation. Finally, we conclude this chapter with a method for approximate circuit search, which was used to generate the datasets later used in the experiments.

3.1 Approximate circuit generation using generative pre-trained model

The closest existing solution to the target of this thesis is a domain-specific generative pre-trained (GPT) model customized for designing approximate circuits (GPTAC) [34]. It uses a modified GPT model trained for node-level vectorization of logical circuits. The circuit design code generated by the model is further fixed and modified by two algorithms. Particularly, the study proposes masking and generalization enhancement algorithms. There is a public GitHub repository where source code and generated circuits are shared. However, no weights are available to test the trained model.

The GPTAC solution aims to be lightweight. It can generate approximate multipliers in less than 15 seconds while consuming only 4GB of GPU VRAM, and the authors claim that the generated multipliers reduce the relative area-to-accuracy by 10% to 40% depending on different accuracy targets.

GPTAC presents a very interesting training and generation process: the model is taught new tokens, and the output is modified while being sampled. A slight downside is the focus on 8-bit signed multipliers rather than exploring performance across different bit widths and generally different circuit types, such as approximate adders.

3.1.1 Architecture

The researchers modified the decoder-only architecture. They did not choose to modify the encoder-decoder because the encoder introduces redundant parameters when the model is given a task like generating circuits, which have short input and long output sequences.

This makes the encoder-decoder architecture inefficient. They reject the standard decoder-only architecture because the circuit constraints instructions are located at the start of the circuit text, and as the prediction window moves, the constraints' impact weakens. This motivated the authors of GPTAC to insert the constraint instructions directly into the moving prediction window and hide the constraints in the final output. This change ensures the constraints stay in context throughout the whole generation process.

3.1.2 Generation

The GPTAC authors proposed two algorithms to modify the output during the process of generating.

The first algorithm is called the mask mechanism. It works by setting the probability of a generated token to zero if it is meant to be an input that has not been defined yet, or when it would cause a syntax error.

The GPTAC authors proposed two ways to improve the generalizability of the generation process results. The first is to replace gates in the pre-output (the previously generated token sequence in the current generation step) with the second-most-probable vectors, which influence further generation and result in diverse outputs. The second approach is named the generalization enhancement algorithm. It uses an approximation set. This set contains approximate counterparts of some cells. Iteratively, it samples a random value from a uniform distribution. If the value is below half of a given threshold, the last generated cell is replaced with its approximation. If the sampled value is between the threshold and half the threshold, the prediction's probability is set to zero, and the resampling continues. In the last case, when the sampled value is bigger than the threshold, the output is not modified. The new cell is accepted only if it does not cause errors.

3.1.3 Dataset and circuit representation

The authors of GPTAC created their own dataset for training their LLM based on EvoApproxLib. They augmented the EvoApproxLib using Versatile Efficiency-Accuracy Configurable Batch Error Estimation (VECBEE) [27] synthesis method. After augmentation, they improved the quality through random exploration, which included random deletion and modification of gates. This dataset is in Verilog format. MSE and circuit area calculated from the circuits were added as prefix information to the circuit data.

They compared training performance across two formats: a logic-expression format using two-input gates and a standard-cell gate-level netlist format. The solutions in this thesis focused on the gate-level circuit format, in contrast to GPTAC, which uses a standard-cell format.

They also compared two tokenization techniques, single-character slicing and circuit-node slicing, shown in Figure 3.1.

The gate-level netlist representation contained redundant, non-essential keywords, which were removed. The paper does not explicitly state which keywords were removed; we assume they removed *input*, *output*, *assign*, and/or other Verilog keywords. They tokenized relevant keywords and designated module and wire keywords as individual tokens. These adjustments improved semantic clarity and shortened LLM input length.

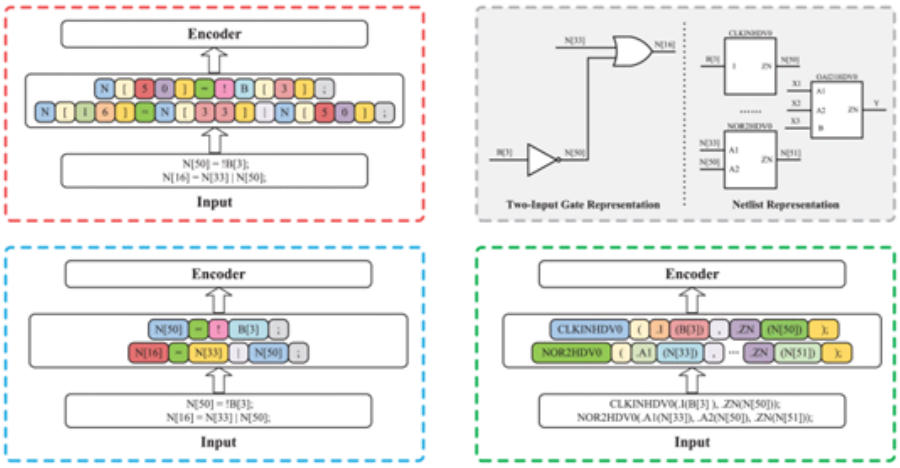


Figure 3.1: GPTAC tokenization process (Top-left: Character-level vectorization; Bottom-left/right: Node-level vectorization for logic-gates/standard-cell-netlist). Taken from [34].

3.1.4 Results

The GPTAC authors conducted experiments comparing formats and tokenization combinations. They used the syntax error rate of the resulting circuits (eq. 3.1) and circuit error rate (eq. 3.2) as evaluation metrics of formats and tokenization methods. The θ denotes the model parameters, N is the number of samples, and x_i is the limiting words. The equations are taken from [34].

$$\text{Syntax error rate} = \frac{1}{N}\sum_{i=1}^{N} P(\text{Syntax Error}|x_i, \theta) \tag{3.1}$$

$$\text{Circuit error rate} = \frac{1}{N}\sum_{i=1}^{N} P(\text{Circuit Error}|x_i, \theta) \tag{3.2}$$

$$\tag{3.3}$$

The results of these experiments are shown in Table 3.1. The table shows that the standard cells format combined with node-based tokenization is the best option for training LLMs to generate approximate circuits.

Table 3.1: GPTAC performance comparison of data formats and tokenization methods of approximate circuits. Adapted from [34].

Data format	**Tokenization**	**Tokens per sample**	**Syntax error rate**	**Circuit error rate**
Two-input logic gate	Single character	26531	62.10%	93.36%
	Circuit node	5192	23.16%	86.72%
Standard cells	Single character	16578	89.32%	31.65%
	Circuit node	3215	31.65%	46.93%

The study compares GPTAC with VECBEE and EvoApprox8b. With different criteria, the performance changes. GPTAC outperformed VECBEE when MSE is 5,000 or higher, excelling with an MSE constraint of 30,000, where it achieved 26.7% MSE improvement and 41% area reduction.

When compared to Evoapprox8b, GPTAC has 7.5% higher latency under the same MSE constraints, but also 9.9% better area. With MSE constraints over 3,000, GPTAC has lower power consumption. GPTAC authors justify these trade-offs as the design priority in low-power and low-area.

3.2 PrefixLLM

LLM-aided Prefix Circuit Design PrefixLLM [33] focuses on optimizing prefix circuits. This method is not specifically designed to search for approximate arithmetic circuits, which are the target of this thesis. Still, this approach is highly relevant to the topic as it shows the direct application of LLM for arithmetic circuit design. Moreover, it shows another way of optimizing circuits with specific target property trade-offs, namely area and delay.

The prefix circuits are fundamental components of digital adders. Their strength is in the efficient calculation of carry signals. The target is to minimize delay and area of the prefix circuits, thereby improving the performance of computing systems. PrefixLLM utilizes LLMs for prefix circuit synthesis.

The efficient computation of carry signals is the most challenging part of the adder design process. The carry computation is a speed bottleneck in adders, as the carry must be computed for each bit of the result. The prefix circuits use a different approach and convert the carry signal computation into generate and propagate signals. These signals are computed in parallel, which reduces the delay.

The prefix circuits are classified by topological configurations, each offering different delay and area trade-offs. There are classical topological configurations such as Sklansky, Kogge-Stone, and Brent-Kung. However, these classical topologies lack the flexibility to adapt to various application-specific constraints. This leads to the need for automatic design methods based on dynamic programming and machine learning.

Even though machine learning methods existed to find suitable prefix circuits, they were based on reinforcement learning (RL). To use RL for finding suitable circuits, circuit synthesis has to be represented as a Markov Decision Process (MDP). This MDP is then solved by RL algorithms such as Deep Q Network (utilized in PrefixRL) or Monte-Carlo Tree Search algorithms. These methods require professional manual design of the state and action spaces. This selection influences the performance of these methods. PrefixLLM shows an effort to overcome the aforementioned constraints of automatic methods.

The PrefixLLM works with structured text. A structured text is an organized text; it must follow certain predefined rules, be in a certain format, or have a defined syntax. It is typically readable by humans, but must be readable by machines. Examples of such texts include VHDL files, Electronic Design Interchange Format (EDIF) files, and, more generally, JSON, XML, or other files. LLMs demonstrated the capability to generate such structured texts. However, a strict structure means errors like misplaced text parts, missing parts, incomplete text, etc., make the generated text unusable. To overcome this challenge, techniques like prompt engineering and iterative refinement in a feedback loop where the output is progressively improved are employed.

To work with circuits, PrefixLLM introduced a new structured text-based representation of the prefix circuit, the Structured Prefix Circuit Representation (SPCR). The task of

synthesizing a valid prefix circuit is defined by the SPCR. SPCR comprises lines where on each line there is a node index, nodes to which this node is connected, a range of the propagate and generate signals, left and right bounds, which are the start and end bit indexes of the compute range. A prefix circuit can be directly constructed from such a description.

An iterative framework generates an SPCR that corresponds to a valid prefix circuit. This framework is captured in Figure 3.2. First, the bit-width n of the target prefix circuit is defined. A partial prefix circuit initialized with n input nodes is iteratively refined by adding a valid prefix node to it until all n output nodes are generated. **Checker** checks the partial prefix circuit in each iteration; if it is valid, the process terminates. Otherwise, it outputs the missing bit ranges of the actual partial prefix circuit. The bit ranges together with the current partial prefix circuit are added to the so-called **SPCR Prompt**, which directs the LLM to fix the partial prefix circuit. LLM generates the **SPCR Response**. The response is in a format where each line contains a node, its predecessor nodes, and a bit range to be added to the current partial prefix circuit. If the response contains invalid prefix nodes, these nodes are removed by the **Pruner** and all valid nodes are added to the partial prefix circuit.

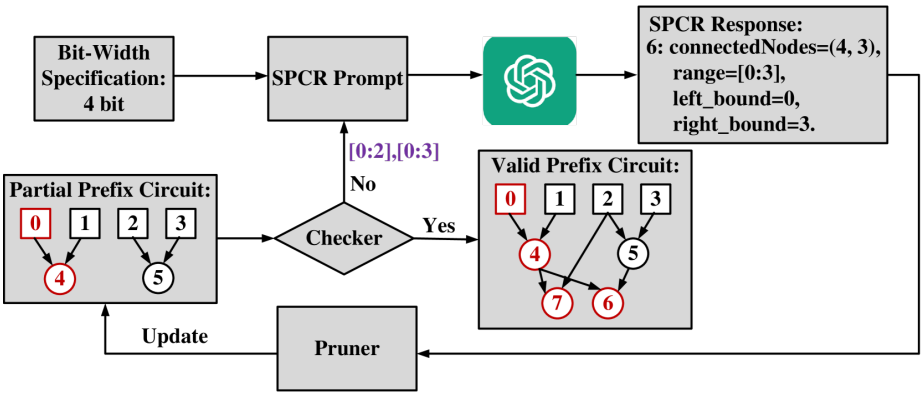


Figure 3.2: PrefixLLM Iterative framework for prefix circuit synthesis. Taken from [33].

Then, the authors of PrefixLLM further extended the iterative framework into the iterative design space exploration (DSE) framework. The DSE framework optimizes the area and delay of the prefix circuit. This framework is shown in Figure 3.3. Specifically, this framework adds area and delay evaluation of the generated circuits. All generated circuits are stored in a sorted prefix circuit pool. This pool is initialized with classical prefix circuits. All circuits are sorted by the non-dominated sorting algorithm by area and delay. In this framework, the SPCR prompt is not fed to LLM, but it is combined with the information from the sorted prefix circuit pool to create a DSE prompt. The DSE prompt is fed to the LLM. The information from the pool helps LLM to generate better SPCR responses that minimize area and delay. The runtime of the DSE framework is limited by the iteration bound. The DSE framework offers modes such as optimizing area given delay bound and minimizing delay given area upper bound.

In the experiments, the authors set the DSE framework to use the OpenAI o1 mini model, and the DSE was limited to 20 iterations. PrefixLLM managed to design 8-bit prefix circuits that across all tested delays show an overall 2.27% node reduction. For 16-bit circuits across all except one delay, it achieved 3.7% node reduction. The reported results for certain delays and bit-widths show better circuits than other existing classical or

automated methods. The authors also compared circuits using Synopsys Design Compiler using Nangate 45nm technology, which further confirmed that the generated circuits are better or at least comparable to other SOTA solutions.

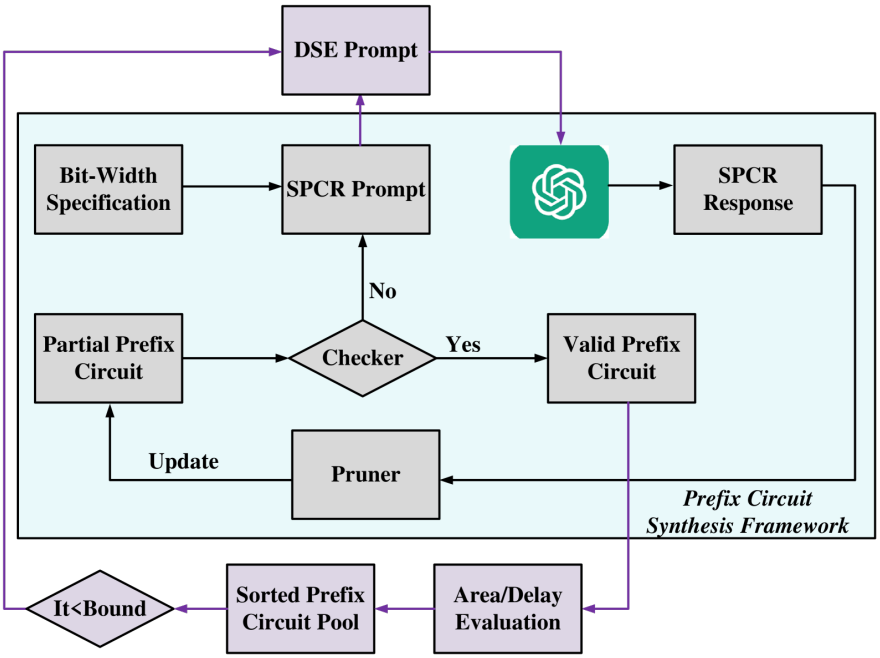


Figure 3.3: PrefixLLM: An iterative design space exploration framework for prefix circuit optimization. Taken from [33].

3.3 Cartesian genetic programming

Another method for automatic circuit approximation is based on multi-objective Cartesian genetic programming (CGP) [11]. The EvoApprox8b and EvoApproxLib datasets used in this thesis are created using this method [21, 23].

The circuits are represented as a two-dimensional array of programmable nodes. This array is called a Cartesian grid (Figure 3.4). The Cartesian grid can be considered an intermediate step towards the phenotype, which is the circuit or a function in general. The Cartesian grid may contain inactive nodes — nodes not on any path from any input to any output — while the circuit contains only the active nodes. Each node in the grid has a function that can depend on the level of abstraction; it can represent a logic gate or other, more complex technology-specific components. The algorithm operates on chromosomes, represented as sequences of numbers. The chromosome is sometimes called the genotype. The chromosome holds information about the Cartesian grid connections, functions, and outputs. New candidate circuits are created by applying small random changes (mutations) to the chromosome. It uses the $(1 + \lambda)$ evolutionary strategy, which means there is one parent, and in the next generation, the candidate is selected from its λ offspring and the parent. The strategy ensures the fitness never decreases [23].

The method typically starts with an exact circuit as illustrated in figure 3.5, where the original circuit is the exact circuit. A mutation operator modifies the circuit, creating

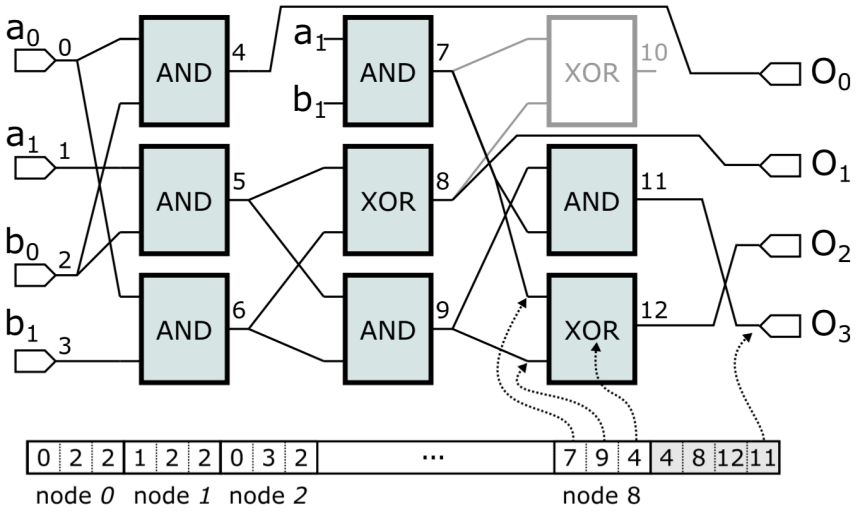


Figure 3.4: CGP representation of a two-bit multiplier. Taken from [23].

new circuits. When optimizing for multiple objectives, such as error and other parameters, areas, or delays, the NSGA-II algorithm [7] manages the circuit population. This algorithm splits the individuals (circuits) into multiple fronts based on the Pareto dominance relation. Then, within each front, individuals are sorted by crowding distance, reducing the number of solutions with similar fitness and promoting diversity. After each iteration of the algorithm, multiple circuits are generated.

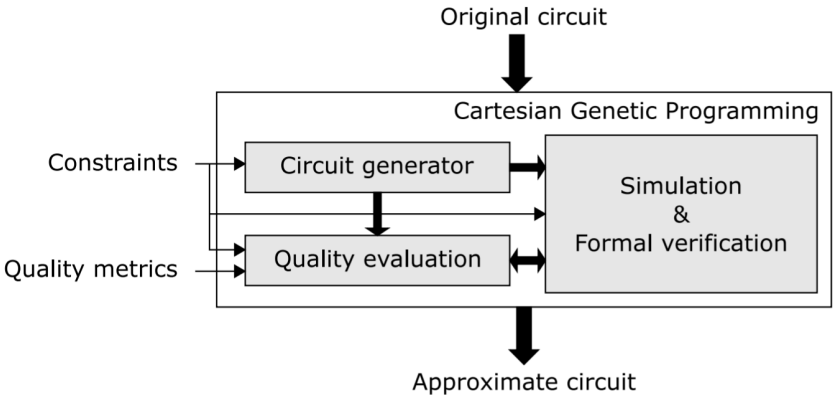


Figure 3.5: Schema of approximation using CGP. Taken from [23].

A fitness function is an important part of evolutionary algorithms, and its selection influences the results. For the fitness function, the authors of EvoApprox8b selected the following fitness objectives: the mean relative error as the error metric, the sum of the areas of components used in the circuit as an area representation, a custom algorithm for power consumption, and the circuit delay is calculated using data from the Liberty timing file available for the utilized semiconductor technology.

3.3.1 EvoApproxLib

EvoApproxLib is a library of approximate circuits. It is an extension of EvoApprox8b. It contains thousands of circuits with various properties. The circuit counts are shown in table 3.2. The library comprises 8 to 128-bit multiplier and adder designs. It provides Python, Matlab, C, and Verilog models for each circuit. There is a publicly available subset called EvoApproxLib$^{\text{LITE}}$.

Table 3.2: The number of components for various circuits in EvoApproxLib. Adapted from [23].

Circuit	**Bit-width**	**# Components**
Adder	8	6,979
	9	332
	12	4,661
	16	1,437
	32	916
	64	176
	128	196
Multiplier	8	29,911
	12	3,495
	16	35,406
	32	349

Chapter 4

Proposed circuit approximation method based on LLM

Based on existing approaches reviewed in the previous chapter, this chapter describes a new method for automated design of approximate circuits with LLMs. We will cover the explanation of circuit representation, the used datasets, the selection of LLMs, how the LLMs will be fine-tuned, and how they will be used for approximate circuit generation.

4.1 Circuit representation

The use of mathematical equations for circuit representation is briefly introduced in Section 2.1.1. For this work, circuits in the boolean equation format at the level of logical gates are used. The circuit is represented as a set of mathematical boolean algebra equations where variables represent nodes of the circuit and the operations are the logical gates (Listing 4.1).

```
N0 = INB7 & INA0
N1 = INB6 & INA1
N3 = N0 | N1
N4 = N0 & N1
N5 = INB5 & INA2
...
N117 = (N105 & N103) | (N116 & N115)
...
OUT15 = ((INA7 & N106) | (N118 & N117))
OUT14 = (N118 ^ N117)
...
OUT2 = 0
OUT1 = N88
```

Listing 4.1: Example Circuit

The main advantage of this format is that the circuit is human-readable and easily comprehensible by pretrained language models, even models that are not trained for this format can at least partially understand it. To test this assumption, we conducted experiments that confirm a certain level of understanding.

GPTAC presented results where they trained an LLM with a similar format, and the model underperformed compared to the netlist format. However, there are two main differences in this work that support the use of the boolean format. The first one is the choice of

LLM. Already pretrained LLMs with significantly more parameters are selected for circuit generation. Moreover, the tested models are already pretrained on datasets comprising mathematical and coding tasks. This pretraining introduces knowledge, which LLMs can exploit in generating circuit designs in the equation format. The second difference is that the format is more condensed by substituting inner nodes into the equation. This could increase the chance of syntactic errors due to mismatched brackets or unfinished lines. On the other hand, it makes the circuit representation shorter, and it may reveal certain patterns in the data during the training phase.

4.2 Datasets

In this thesis, we used EvoApproxLib, an open-source library of approximate circuits, and EvoApprox8b, an older version of it. In addition to the publicly available data, we use an extended version of the current EvoApproxLib.

The first version, also called EvoApprox8b, is available in an archived GitHub repository[1]. This contains 8-bit unsigned adders and multipliers. The circuits are available in different formats: Verilog, Verilog PDK45, C, and Matlab. All circuits also have their parameters written in the code comments. This dataset is used in the first few-shot measurements where the model types are evaluated.

The current EvoApproxLib$^{\text{LITE}}$ version is available online[2]. This version offers circuits of different types, including 8-, 12-, and 16-bit signed and unsigned adders; 8-, 12-, and 16-bit signed and unsigned multipliers; 7- and 11-bit unsigned multipliers; 8×2, 8×3,...8×7-bit unsigned multipliers; and more. The circuits are available in multiple formats like Verilog, C, and Python. On the website, users can select a Pareto optimal subset of different criteria, like MRE vs power. The only downside of this dataset is that the full version is not publicly available yet. This dataset is used for testing how LLM behaves when prompted to generate circuits of different bit widths.

Finally, a subset of the full EvoApproxLib is used. This set contains 25k 8-bit unsigned approximate multipliers. We further split it into train, validation, and test subsets. These subsets are used for training LLMs and evaluating their performance.

4.3 Large language models

We targeted the LLMs with 7B parameters. These LLMs are relatively large compared to GPTAC, while still possible to run on consumer-grade hardware. To find the most suitable model, we selected a representative LLM from four types, which we consider relevant to circuit generation. We searched the Open LLM Leaderboard[3] to find suitable candidates. Here, I targeted 7B models from official providers. The search resulted in the selection of the following models: Qwen2.5-Math-7B, OpenReasoning-Nemotron-7B, Qwen2.5-Coder-7B-Instruct, and Falcon3-7B-Instruct.

Falcon3-7B-Instruct[4] represents general LLMs. This model performed best on the leaderboard, meeting the filter criteria of parameter count and only official providers. This model was trained on 14 Teratokens of datasets. It is worth noting that the code and STEM

[1] https://github.com/ehw-fit/evoapprox8b/tree/master
[2] https://ehw.fit.vutbr.cz/evoapproxlib/
[3] https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard
[4] https://huggingface.co/tiiuae/Falcon3-7B-Instruct

data were included in the training dataset, potentially introducing relevant knowledge for the circuit generation task.

Reasoning models improve the final output by iterating over their own outputs. OpenReasoning-Nemotron-7B[5] is the selected representative of the reasoning models. It is post-trained for reasoning about science solutions, math, and code. This model is based on Qwen2.5-7B.

There are models that are trained to solve math problems. One of the best such models is Qwen2.5-Math-7B [6]. Fine-tuning on a dataset containing math problems introduces knowledge, which this model can utilize in understanding the boolean equation format.

I also included a model whose fine-tuning aimed for code generation. This is because the circuit description can be viewed as a program code, and the models trained on multiple programming languages can better understand the format in which the circuit is written. As code model representative, I have chosen Qwen2.5-Coder-7B-Instruct[7]. This model is post-trained on code as well as on math datasets.

4.4 Measurements

When an LLM generates a circuit design, the parameters of the circuit need to be measured to compare the circuits effectively. Errors are measured by conversion to C code. The code is run and tested on all input-output pairs. During this process, the MSE and MAE errors as defined in Section 2.1.4 are measured.

The area is calculated using the gate sizes in μm^2 for the 45nm technology. All areas are summed together, and this single number is the area estimate of a physical circuit.

For circuit structure comparison, a metric based on graph edit distance is created. The equation format contains long expressions with nested parentheses, making the task of direct graph generation difficult. To compare two circuits, each circuit must be converted into a simpler format. The processing starts with a circuit in the equation format. The circuit must be processed by a custom recursive descent parser. The parser builds a parse tree. From the tree, rows containing a node with one assignment and at most one operation between two other nodes are created recursively. From this format, it is trivial to build a directed acyclic graph structure in any library implementing graph algorithms.

4.5 Few-shot prompting

In the first experiments (Section 6.1), it is important to find out how selected LLMs perform on the chosen dataset and with the data transformed to the selected format. The only reasonable option here is to test one-shot and few-shot prompting [2]. A circuit is generated given a specific prompt. This prompt contains target circuit requirements, one or more examples of circuits in the required format, and parameters of the example circuits.

Using this few-shot prompting method, I first compare the performance of the selected models. The comparison will be done by constructing a prompt with two circuits. The prompt will require the model to minimize MSE while preserving area. The generated output will be tested on a number of syntactically correct circuits, and the circuits that fall

[5] https://huggingface.co/nvidia/OpenReasoning-Nemotron-7B
[6] https://huggingface.co/Qwen/Qwen2.5-Math-7B
[7] https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct

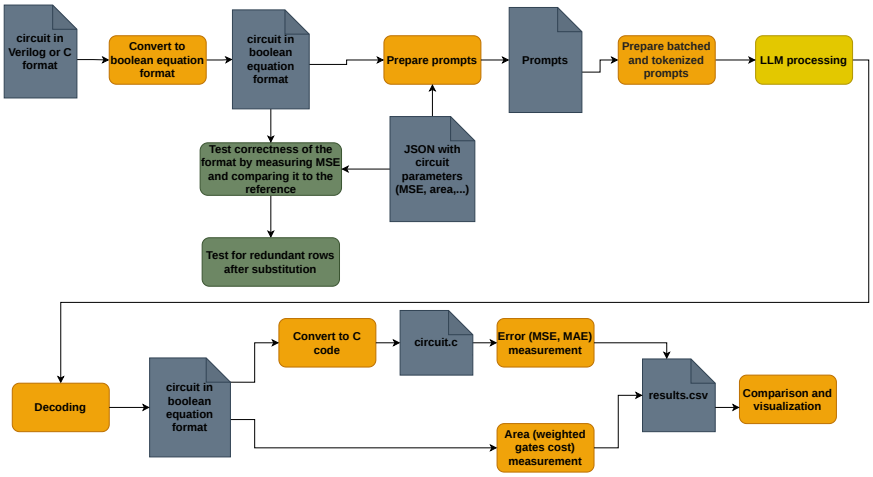


Figure 4.1: Few-shot pipeline with format processing.

into the range of target error and area. The syntax is tested by converting the design into C code and running it during error measurement.

When the best model for few-shot prompting is found, further experiments are executed. The following experiment aims to find out how many example circuits in the prompt are the most suitable. It is described in Section 6.2.

The pipeline for the few-shot prompting is shown in Figure 4.1. Circuits from Verilog or C files are first converted to the boolean equations. This format is tested for correctness compared to the given circuit parameters. The testing is represented by the green squares. By this testing the format converting script is tested. Prompts are prepared from circuits in equation format and from the parameters of these circuits, which are stored in a JSON file. These prompts are grouped and fed to the LLM in batches. The LLM output is decoded and measured. The measurement is done in two parallel branches. First one converts the generated circuit to C code and measures errors like MSE and MAE. The second branch works with the equation format and calculates the area as the weighted gate costs summation. The results of measurements are stored in a CSV file. The measured properties of the generated and target circuits are then compared and visualized.

4.6 Training

Based on experiments with few-shot prompting, we fine-tune a 7B model, Qwen2.5-Coder-7B-Instruct, which performed best among the tested LLMs. To improve the model's understanding and reduce the fine-tuning time, we added new tokens to its vocabulary. The new tokens represent the circuit nodes.

To make the fine-tuning efficient and possible on smaller GPUs, we use Quantized Low-Rank Adaptation (QLoRA) [8]. QLoRA stems from LoRA [12]. The LoRA freezes weight matrices W_0 at each layer of the model. It adds the low-rank matrices A and B to each such weight matrix. These matrices are updated during training $W_0 + \Delta W = W_0 + BA$. The matrices have dimensions $W_0 \in \mathbb{R}^{d \times k}$, $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$. The symbol r denotes rank; it is smaller than the minimum of the dimensions d and k. When the matrices are trained, the forward pass is performed by summing the multiplication of the previous layer output with matrices and summing the results together $W_0 x$ and BAx, as shown in the

equation: $h = W_0 + \Delta W x = W_0 x + BAx$. When training with LoRA, one often decides the rank size and the alpha parameters. The alpha is a scaling factor that determines the influence of the learned adapter weights. Mathematically written $\Delta W x$ is scaled by $\frac{\alpha}{r}$. The QLoRA keeps the AB matrices in high precision but uses the base model quantized to a new datatype, 4-bit Normal Float (NF4). The NF4 is based on quantile quantization, ensuring that each part of the data receives an equal number of values from the input tensor. QLoRA dequantizes (the inverse of quantization) the weight tensor before each use, converting it to BFloat16 precision.

We train the models on a dataset derived from EvoApproxLib. We prepare prompt templates and insert the data from EvoApproxLib into them.

4.7 Layer inversion

This thesis is about using LLM to find solutions to a multi-objective problem, in other words, to find something new, the circuits. Even though there are various ideas, such as adding symbolic systems and changing LLM temperature to increase creativity, we were motivated by an article[8] and studies [35, 25] to conduct a few experiments related to this topic[9]. But our goal was to increase the diversity of the generated output, leading to better design state-space exploration.

Temperature introduces a randomness factor into the generation process and increases the diversity. But this only influences the model's final output in the decoding phase, not the information processing in its inner layers. The inner layers of the model can be modified, and there is a technique called steering vectors [28], but these vectors must be learned to output a specific tone in texts.

A more direct approach is to apply a transformation to the output of certain layers directly. We conducted smaller experiments with increasing values of the hidden state vector near zero, or pushing activations in different directions from the mean. Then we came up with equations 4.1–4.2.

$$h'[i] = \frac{1}{\text{clamp}(|h[i]|, 10^{-5})} \tag{4.1}$$

$$h' = \frac{h'}{\max(h')} \tag{4.2}$$

where h, h' are the original and new hidden state respectively;

$\text{clamp}(a, b)$ when a is smaller than b returns b, otherwise a; prevents division by zero;

$\max(v)$ selects element of given vector v with the highest value.

The essential idea behind this formula is to make the activations near zero (consider them „hesitant“) the strongest, and vice versa. So the most activated ones are set to small values. Indecisive and unlikely inner states („ideas“) are promoted. The absolute value in the equation makes even the negative values around zero strongly positive, making them

[8] https://fortune.com/2025/06/20/hugging-face-thomas-wolf-ai-yes-men-on-servers-no-scientific-breakthroughs/

[9] During designing the experiments, I read an article about a statement of Thomas Wolf, the CEO of Hugging Face, claiming that LLMs are unlikely to make scientific breakthroughs. First, I did not pay much attention to it, but also studies like [35, 25] show that LLM has been limited in its impact in producing novel, expert-level ideas and performing discovery of new principles or new scientific laws.

likely to activate, while keeping the very negative values small. The maximum of new values normalizes the values to match the maximum of the original values. This step is crucial to keep the activations in bounds.

The formula needs to be applied to the layer with the most semantic meaning, so it can influence the logic, while not disturbing the syntax. For this purpose, we suggest searching for this layer throughout the whole model.

As mentioned before, this approach resembles steering vectors in influencing the inner layers of LLMs. But it diverges in two aspects. First, the steering vector is learned; the proposed approach does not incorporate learning any parameters. Secondly, the steering vector guides LLM to specific responses, while the layer inversion promotes diversity of the output.

4.8 Refinement loop

Generating a circuit in a single LLM pass results in many failures just due to minor syntactic errors and also minor or major logical flaws. Implementing the refinement loop [20] is a technique to mitigate these sorts of errors. The algorithm is modified to support circuit evaluation, which provides better responses in the form of measured metrics and program runtime error reporting. The proposed modified algorithm is shown in Figure 4.2.

The algorithm starts by prompting a fine-tuned LLM to generate a circuit design with the required parameters, with no initial circuit example. The second and subsequent iteration prompts include one example of a circuit generated in previous iterations. Each iteration generates a single circuit for a single target. Circuit parameters give the target. It searches for multiple target circuits in parallel to utilize the GPU fully via parallel inference. In the second and subsequent iterations, a new prompt is formulated incorporating the best-so-far solution (Algorithm 2) and the last valid solution (Algorithm 1), along with its measured parameters or a described error. The description of the compilation error from circuit measurement proved ineffective in early small-scale experiments, so we decided to remove it from the algorithms.

When the algorithm in one iteration generates more than a certain number of circuits that do not improve the best solution or are faulty, it indicates a need for change. The change is done by raising the temperature. The algorithm ends after the predefined number of iterations is reached.

4.9 Evolutionary algorithms

Evolution through large models (ELM) [18] uses LLM as an intelligent mutation operator. It can be combined with MAP-Elites, a quality-diversity algorithm that stores only the high-performing „elite“ individuals, which differ in meaningful ways.

After testing the refinement loop, this approach is an improvement. To take advantage of the evolution, we implement two types of loops — one (Evolution with LLM), where a fine-tuned model iteratively modifies a population of circuits, and the second, with larger pretrained LLMs, using circuits and prompts for LLM coevolution. The initial circuits used in these algorithms are from EvoApproxLib. The reason for this choice is that pretrained models do not know the equation format, and the model in the simpler evolution is specifically fine-tuned for modifications rather than generating from scratch.

Algorithm 1 Last valid refinement loop.

Require: Initial LLM LLM_{init}, Refinement $LLML_{refine}$, prompts P, data D, settnings S, max iterations I, worse circuits threshold W

```
Initialize temperature
Initialize worse counter w ← 0
B ← [None]
seed ← random()
for i = 0 to I − 1 do
    if w exceeds threshold W then
        Increase temperature in S
    end if
    if i = 0 then
        C ← L_init(P_init, S, seed)
    else
        C ← L_refine(P_refine, S, seed)
    end if
    E ← Evaluate(C)                                          ▷ Evaluate circuits
    w ← 0                                                    ▷ Reset counter
    for each circuit i do
        if E_i is valid then
            if B_i is None or (E_i.area ≤B_i.area and E_i.error ≤B_i.error) then
                B_i ← E_i                                    ▷ Update best circuit
            else
                w ← w + 1
            end if
            V_i ← E_i                                        ▷ Update last valid circuit
        else
            w ← w + 1
        end if
        if V_i is None then
            Keep original prompt P_i
        else
            Update prompt P_i using V_i      ▷ Use last valid circuit as example in next iteration
        end if
    end for
    Update seed
end for
Store final, best, and evaluation results in D
return D
```

Algorithm 2 Hill climbing refinement loop.

Require: Initial LLM LLM_{init}, Refinement $LLML_{refine}$, prompts P, data D, settnings S, max iterations I, worse circuits threshold W

```
Initialize temperature
Initialize worse counter w ← 0
B ← [None]
seed ← random()
improves ← (E_i.area ≤ B_i.area) and (E_i.error ≤ B_i.error)
for i = 0 to I − 1 do
    if w exceeds threshold W then
        Increase temperature in S
    end if
    if i = 0 then
        C ← L_init(P_init, S, seed)
    else
        C ← L_refine(P_refine, S, seed)
    end if
    E ← Evaluate(C)                                   ▷ Evaluate circuits
    w ← 0                                             ▷ Reset counter
    for each circuit i do
        if (E_i is valid) and improves then
            B_i ← E_i                                 ▷ Update best circuit
        else
            w ← w + 1
        end if
        if B_i is None then
            Keep original prompt P_i
        else
            Update prompt P_i using B_i ▷ Use best circuit as an example in next iteration
        end if
    end for
    Update seed
end for
Store final, best, and evaluation results in D
return D
```

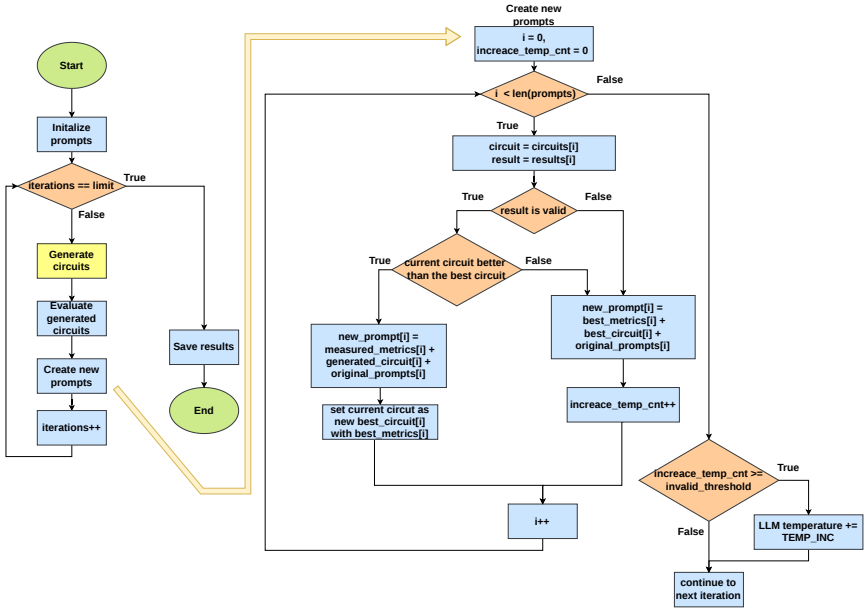


Figure 4.2: A diagram of an algorithm for automatic loop refinement of generated circuits.

Evolution with LLM starts with a population of circuits. The model is tasked with improving one metric while not drastically worsening the others. The LLM performs a form of mutation. The algorithm then selects a new population from the parents and the offspring. For the selection, it uses NSGA-II survival strategy with Pruning Crowding Distance [16]. The fitness functions are F_1^C (WCE or MSE) and F_2^C (area). The LLM is asked to improve one of the circuit metrics, which are identical to the fitness functions. For that purpose, the algorithm inserts a metric based on the fitness functions into the prompt template 5.4. Each circuit in the population is mutated for each fitness, so the LLM is called twice the number of circuits per generation. The algorithm ends after reaching a specified number of generations. This algorithm is similar to the evolution of circuits (Section 4.10.1) in the following coevolutionary algorithm.

The second algorithm (Coevolution) uses a pretrained models without further fine-tuning, with circuit and prompt coevolution as in Figure 4.3. The evolved prompts (Figure 4.4) will guide the LLM toward finding the final solution, and the generated circuits (Figure 4.5) will improve as it finds new prompts. The initial prompt population contains handcrafted prompts that encourage the model to perform operations such as removing, replacing, or inserting parts of the circuit, or to improve the source circuit based on other circuit examples. As a fitness function, the metrics described in Section 4.4 are utilized. A fine-tuned or code-pretrained LLM acts as a mutation operator for the circuits, while, for the prompts, a general-purpose LLM can be used. The prompts are evaluated by the difference between circuits before and after the mutation. So, the prompts are assigned scores based on the circuit metrics, or penalized when a faulty circuit is generated. Both the circuits and the prompts form Pareto fronts. After reaching a predefined number of iterations, the algorithm is terminated.

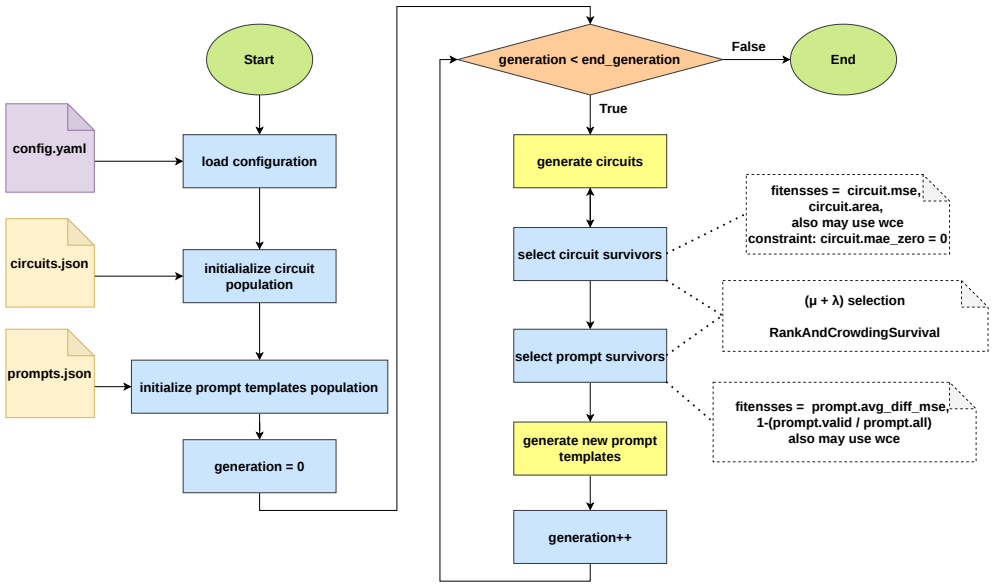


Figure 4.3: A high-level diagram of the prompt and circuit coevolution algorithm.

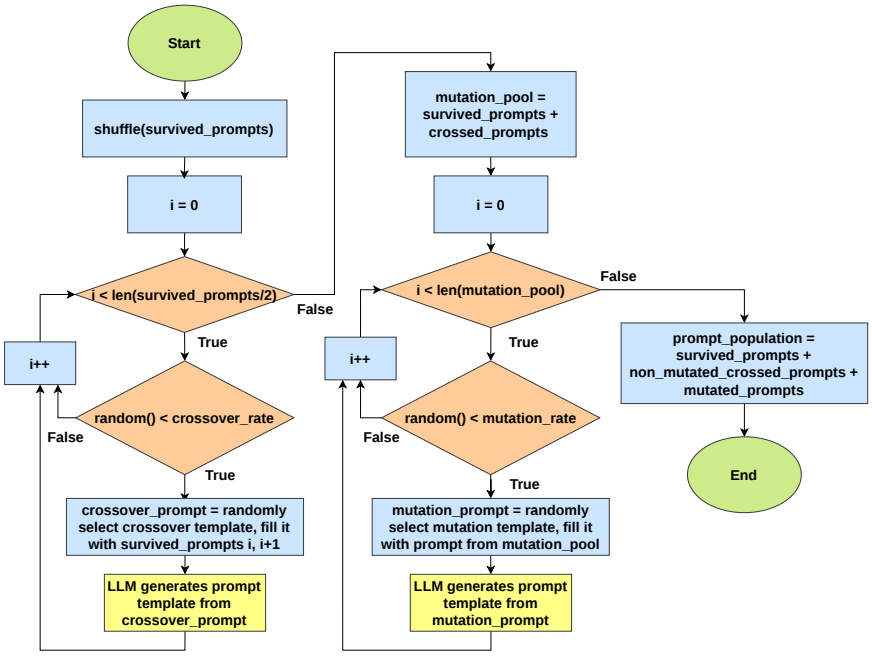


Figure 4.4: A diagram of the prompt crossover and mutation algorithm as part of coevolution through a language model.

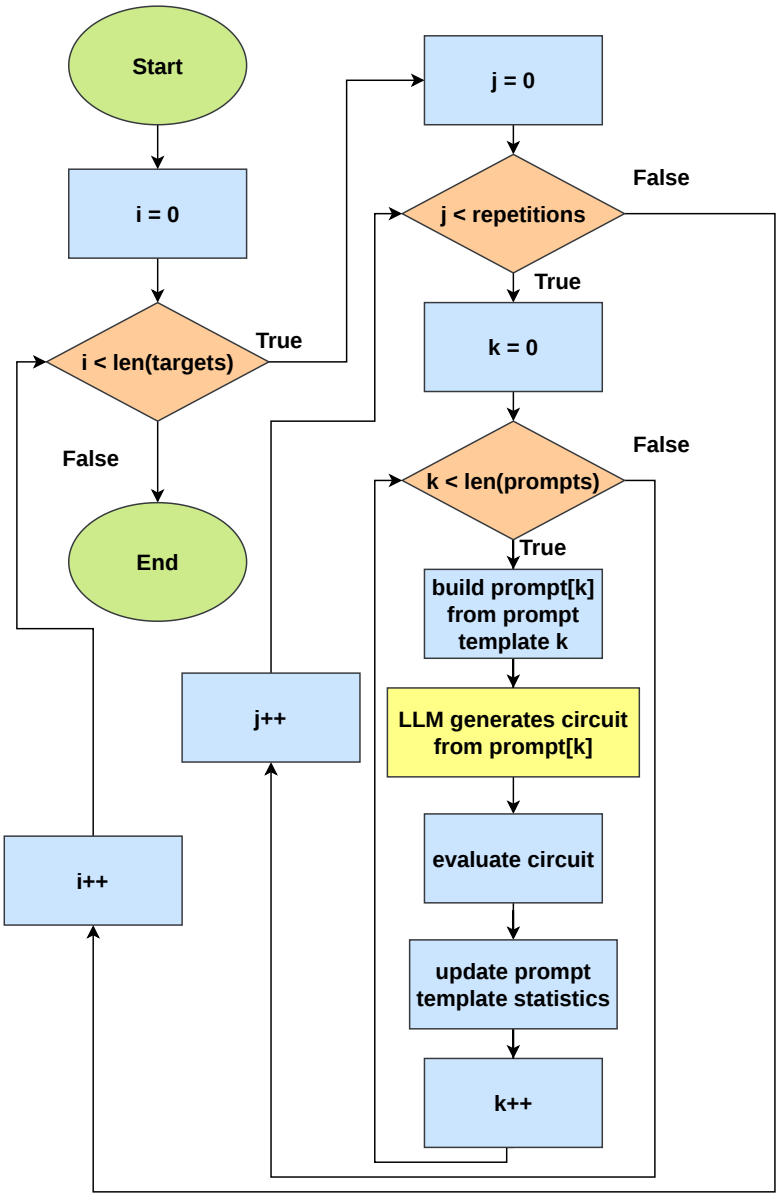


Figure 4.5: A diagram of the circuit generation algorithm as part of the coevolution through the language model.

4.10 Coevolution

Coevolutionary algorithms (CoEAs) are classified according to how individuals interact with one another. When individuals are evaluated by interacting with other individuals in the same population, the algorithm is a single-population CoEA. On the other hand, in multi-population CoEAs, individuals in one population interact with individuals in one or more other populations [10]. We use a multi-population CoEA. The algorithm maintains populations of circuit designs and LLM prompt templates.

The algorithm's objective is to generate an approximate multiplier that closely matches the user-provided target error T_E and area T_A. As the exact target values are usually unattainable, we designed the algorithm to solve a multi-objective optimization problem. It builds a set of multipliers with error and area on the Pareto front near (T_E, T_A).

Figure 4.6 shows the scheme of the method whose main idea is described in this section. Details about its components and implementation are provided in the following chapter. The method co-evolves two populations: (1) a population of circuits and (2) a population of circuit-producing templates (templates, for short). In the evolution of circuits, an LLM is used as a mutation operator to generate a new candidate circuit from one or more circuits in the current circuit population. The selection depends on the template. New circuits are created using templates that are cultivated in the second population. Templates help to create strong prompts for LLM-based circuit modification. Their adaptation is essential as it is not always clear how to instruct LLM to build a useful offspring circuit.

4.10.1 Evolution of circuits

A candidate multiplier is represented in the format described in Section 4.1

The initial population is seeded with approximate 8-bit multipliers of various quality.

Fitness $F_1^C(c)$ is the error of a candidate circuit c determined according to Eq. 2.3 or 2.1 (depending on the objectives) for all 2^{16} input vectors. In addition, it is strictly requested that the result of multiplication by zero is zero. In other words, the circuit c performs the exact multiplication if at least one of the operands is zero. If this condition does not hold, the candidate circuit c is considered invalid for selection into the next population of circuits. This requirement is critical for efficient deployment of approximate multipliers in DNN accelerators [22]. The second fitness, $F_2^C(c)$, is the area of c on a chip, which is estimated with respect to the target fabrication technology as the sum of the sizes of all gates involved in circuit c. The fitness values F_1^C and F_2^C are stored in a data class dedicated to the circuit code.

LLM-based Operators: A new candidate circuit is generated by LLM using a prompt created according a template, e.g. Listing 4.2.

```
Given a circuit with these parameters:
Worst case absolute error (WCE) = {approx_circuit_wce}
Area = {approx_circuit_area} um^2
The circuit: {approx_circuit}
Generate a new 8-bit unsigned approximate multiplier circuit with parameters:
Worst case absolute error (WCE) = {target_wce}
Area = {target_area} um^2
Generate only the new circuit, without any comments.
The new circuit:
```

Listing 4.2: Example Template

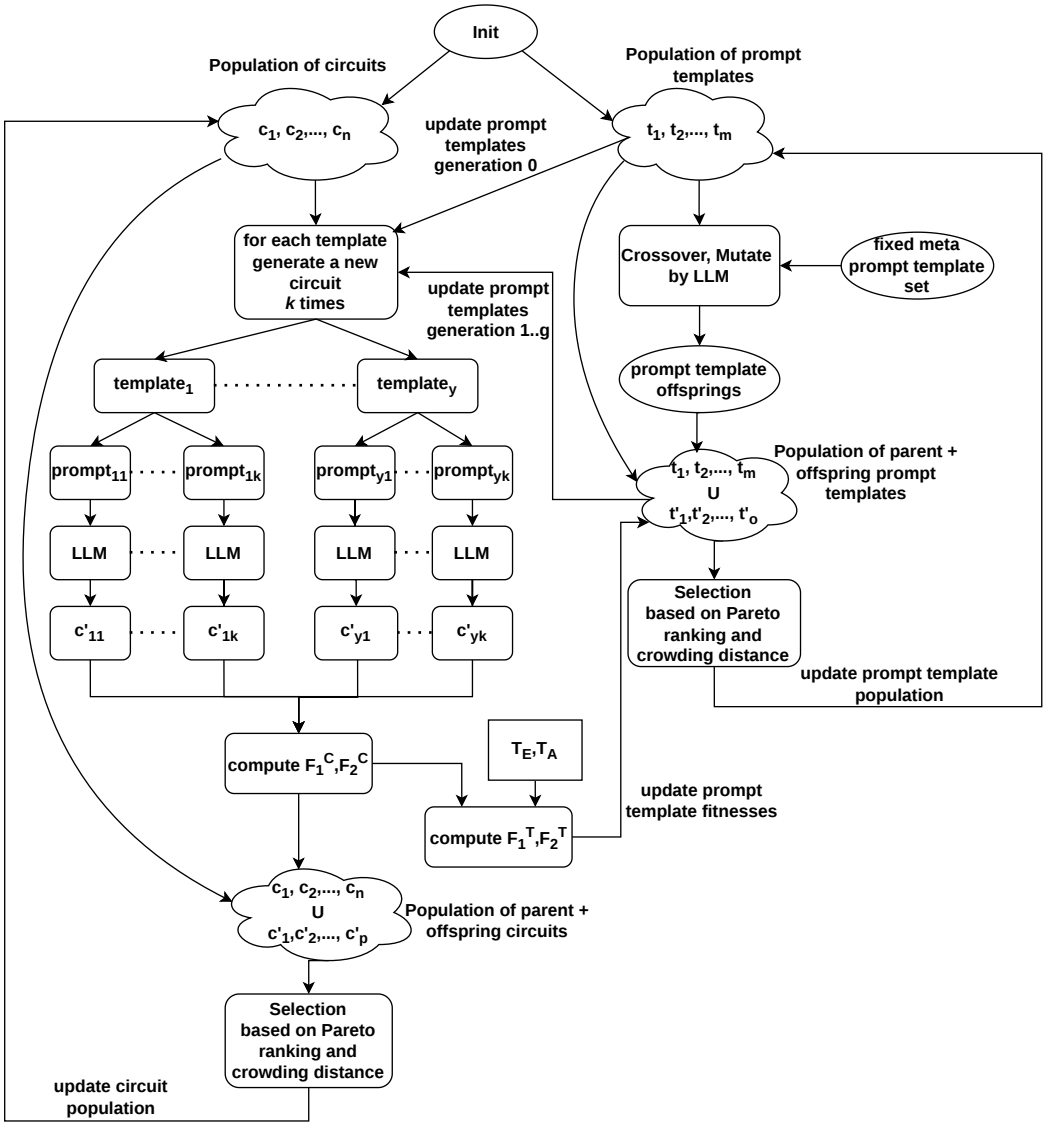


Figure 4.6: The proposed circuit approximation method that utilizes a coevolutionary algorithm evolving a population of circuits (left-hand side) and a population of circuit- producing templates (right-hand side). The circuit- producing templates population provides templates for generating new circuits. After the circuit fitness measurement, template fitness functions are measured and used by the template population.

Templates contain placeholders determined by {...} for circuit parameters (errors, area) and the circuit description itself (a particular circuit description is given in Example Circuit 4.1). A template can refer to various existing circuits of the current population using the following identifiers: `random`, `closest to target`, `second closest to target`, a circuit with the `best error metric`, or a circuit with `the best area`. Circuits corresponding to these identifiers are stored and updated in local memory and can be used during the co-evolution. In a given generation, the algorithm employs up to y templates, which can be reused. For a given circuit c_i and a selected template, LLM is executed k times to create k offspring circuits (see the left loop of Figure 4.6).

The sets of parents and offspring are combined, and the algorithm selects the new population from this union. The selection is based on the NSGA-II [7] survival strategy.

4.10.2 Evolution of templates

The evolution of templates is visualized in the right-hand loop of Figure 4.6.

A template is a blueprint for prompt creation. It contains immutable text and placeholders as illustrated in Example Template 4.10.1. The placeholders are dynamically replaced with specific data.

The initial template population includes manually created templates for creating prompts. These prompts instruct the LLM to generate new candidate circuits in the first generation of circuit evolution. All templates are stored in an external JSON file. The initial circuit-producing templates are called Generate_prompts. The JSON file also contains templates for modifying circuit-producing templates. These are Mutation_prompts, Crossover_prompts, Post_process_prompt_circuit and Post_process_ prompt_prompt.

Fitness $F_1^T(t)$ of a template t is defined as the ratio of valid and invalid candidate circuits generated by LLM using t during one iteration of the circuit evolution loop. More precisely we define the $F_1^T(t)$ by equation 4.3. We apply the term $1 - x$ to convert a maximization objective into a minimization objective. The *max* function in the denominator prevents division by zero. Fitness $F_2^T(t)$ of a template t is defined as the average difference between target error T_E and the errors of candidate circuits generated using template t. Each circuit-producing template t is stored in a data class together with its fitness values $F_1^T(t)$ and $F_2^T(t)$.

$$1.0 - \left(\frac{prompt.valid\_cnt}{\max(prompt.circuit\_cnt, 1)} \right) \tag{4.3}$$

Template-modifying operators are again LLM-based. A set of prompt templates is specified in advance for the mutation and crossover of circuit-producing templates. A template-producing template is randomly selected during the genetic operation. The template-producing templates contain a fixed text with instructions for LLM, variables like `{valid_count}` denoting the number of valid circuits generated using this template, and particular template(s) for generating circuits. Examples of prompts are in Listings 4.3 and 4.4.

The selection of circuit-producing templates is similar to circuit selection. The sets of parental and offspring circuit-producing templates are united, and from this union, a function based on the NSGA-II survival strategy composes the new population.

```
You are given a prompt template: "{prompt_template}"
This template generated {valid_count} valid
and {invalid_count} invalid circuits.
Modify the template; the new template should guide the large language model to
    generate circuits more precisely, given target parameters. Generate only the
    new template, without any comments. Output the new template:
```

Listing 4.3: Example prompt for mutation

```
You are given two prompt templates
Template 1 with {valid_count1} valid and {invalid_count1} invalid circuits:
    "{prompt_template1}"
Template 2 with {valid_count2} valid and {invalid_count2} invalid circuits:
    "{prompt_template2}"
Create a new prompt template by combining the two prompt templates. The new
    templates should steer large language models to generate circuits with a
    smaller area while keeping WCE the same or lower. Generate only the new
    template, without any comments. Output the new template:
```

Listing 4.4: Example prompt for crossover

4.10.3 Co-evolutionary mechanisms

The two evolutionary loops are synchronized every generation. For each template t_i, fitness $F_1^T(t_i)$ and $F_2^T(t_i)$ are computed after evaluation of all circuits generated from the template t_i using F_1^C and F_2^C. Coevolution terminates after g generations. The coevolutionary algorithm produces a set of circuits that occupy a Pareto front (error vs. area) close to target T_E and T_A.

4.11 Circuit types

In the experiments, we focused on unsigned 8-bit multipliers. There is a valid question of whether systems built to generate one type of circuit can be reused to generate different types of circuits, such as 7-, 11-, and 12-bit multipliers or even adders of various bit widths. To answer this question, a separate set of experiments needs to be conducted. The experiments will use the same systems described in this chapter.

It is assumed that, before introducing the refinement loop or changing the training to include these new circuit types, the systems will not perform well, if they even produce meaningful output. However, specifically designed prompts, whether evolved or developed through an iterative process, may be the key to greater generality.

Table 4.1: Sizes of the gates in μm^2 corresponding to the 45 nm technology.

Gate	INV	AND	OR	XOR	NAND	NOR	XNOR
Size	1.40	2.34	2.34	4.69	1.87	2.34	4.69

Chapter 5

Implementation

The proposed solutions are implemented primarily in Python, using libraries for large language models (PyTorch, Transformers), evolutionary algorithms (pymoo [3]), and others. Circuit measurements are written in the C language to increase evaluation speed.

5.1 Format conversion

Source circuits are available in multiple formats in the EvoApproxLib. We can use the C language and Verilog format and convert them to the boolean equation format presented in Section 4.1. For this purpose, we created Python scripts. The scripts utilize regular expressions to extract equations from the code.

Script *c_to_equations.py* processes each circuit in the dataset as follows. It cleans the C circuit description of C-specific keywords. Inputs, outputs, and auxiliary variables are identified and stored in the corresponding lists. The script iteratively identifies nodes that appear exactly once on the right-hand side across all expressions, and inserts their definitions into the expression instead of the node name. This makes the circuit description shorter. In the end, it orders the remaining intermediate nodes and renumbers them starting from zero, keeping the original order.

5.2 Circuit error and area computation

In all experiments, we need to calculate the errors MSE, WCE, MAE, and MAE for multiplication by zero. This operation needs to be fast to avoid slowing down the algorithms. We decided to implement it in C using the OpenMP library. For further acceleration, we also precompute the 256×256 multiplication table for input combinations and store it in binary form. The code is in the file *errors.c*; from Python, we call it using the subprocess library.

Area is estimated in the function *circuit_gate_costs()* using the *re* (regular expressions) library. It counts the number of each gate type, multiplies the per-gate counts by the values from table 4.1, and sums the results to obtain an overall area estimate.

Algorithms that call the LLM in a loop use the Python script *compute_errors.py*. It computes errors in separate threads. First, in each thread, it converts the circuit from equation format to C code, then compiles and runs the *errors.c* subprocess. Each thread estimates the area of its circuit after the error computation. To avoid errors due to simultaneous execution, binaries have unique names per thread. The error measurement also

serves as a partial validity checker. When the circuit's C code can not be compiled, it signals an invalid circuit design.

5.3 Dataset preparation

We fine-tune models on a dataset based on the EvoApproxLib. We prepare prompts that include circuits and their parameters, and refer to this set as the fine-tuning dataset. Circuits in the prompts are in the equation format. For model fine-tuning, it is a good practice to split the dataset into training, validation, and test splits. The source EvoApproxLib dataset consists of 24,912 8-bit unsigned multiplier circuits. We kept most of the circuits for the training split. 250 circuits are selected for the validation, and 250 for the test dataset splits. The remaining circuits are in the train split, totaling approximately 24.5k circuits. The circuits are selected so that the test split contains Pareto or near-Pareto circuits for the WCE/area and MSE/area objectives. The validation split contains only non-Pareto circuits. The train split contains a mix of circuits of different quality.

The data from EvoApproxLib is inserted into predefined prompt templates, allowing us to create prompts. A prompt is the fundamental piece of the dataset for fine-tuning.

LLM distinguishes between two roles: user and assistant. During fine-tuning, LLM uses the user part of the prompt for the forward pass. LLM generates output and, using the cross-entropy loss function, computes the error between the output and the assistant prompt part.

We use three types of prompt templates—the first (*Train_generate_template*), displayed in Listing 5.1. The user part contains the target circuit parameters, and the assistant part contains the corresponding circuit design. The parameters are inserted into each prompt in a shuffled order. Each line in the Listing 5.2 shows a parameter template; shuffling changes the order of the lines. The shuffling prevents the LLM from relying on parameter order during inference.

```
Given the following parameters: {params_text}
Generate the corresponding 8-bit unsigned approximate multiplier circuit:
```

Listing 5.1: Example prompt template for fine-tuning, *Train_generate_template*.

```
Mean Squared Error (MSE) = {mse}
Mean absolute error (MAE) = {mae}
Mean absolute error for multiplication by zero = {mae_zero}
Worst case absolute error (WCE) = {wce}
Synthesised circuit size = {pdk45_area}
Area = {circuit_gate_cost(circuit)}
Delay = {pdk45_delay}
Electric energy consumption = {pdk45_pwr}
```

Listing 5.2: Set of parameter templates.

The second prompt template type (*Train_improvement_with_target_template*), in Listing 5.3, is similar to the first type. The difference is that the prompt starts with an example. The example consists of circuit parameters (MSE, WCE, area, etc.) and a corresponding circuit design. After the example, the requested circuit's parameters are provided. The pairs (example and reference target circuit) for this type of prompt are selected by the *find_high_impact_neighbors* to minimize edit distance and maximize the difference in metric values (errors and area).

```
Given a circuit with these parameters: {src_params_text}
The circuit:
{circuit_src}
Generate a new 8-bit unsigned approximate multiplier circuit with parameters:
    {tgt_params_text}
The new circuit:
```

Listing 5.3: Example prompt template for fine-tuning with improvement over given circuit and target, Train_improvement_with_target_template.

The third prompt template type (*Train_metric_improvement_template*), as listed in 5.4, starts with an example similar to *Train_improvement_with_target_template*. However, it does not specify target circuit parameters after the example. Instead, it commands the LLM to generate a circuit with a lower error metric or a smaller area. In the dataset, the second circuit must improve on at least one selected metric, while keeping the others relatively the same (better or just slightly worse). The circuit pairs for this prompt type are selected similarly to those for the *Train_improvement_with_target_template*. However, here we created a function *find_neighbors*, in which we tuned thresholds for minimal and maximal metric improvements and set a limit on the maximal edit distance. The pair counts in the dataset differ per metric. So we leverage greater control to generate sets of similar size for each improvement goal (e.g., improve MSE, improve area, etc.) by tuning the thresholds.

```
Given a circuit with these parameters: {src_params_text}
The circuit:
{circuit_src}
Generate a new 8-bit unsigned approximate multiplier circuit with lower {metric}.
The new circuit:
```

Listing 5.4: Example prompt template for fine-tuning with improvement over given circuit and metric to improve, Train_metric_improvement_template.

5.4 Fine-tuning

We fine-tuned Qwen2.5-Coder-7B-Instruct model on the boolean equation format introduced in section 4.1. For the fine-tuning, we use the *SFTTrainer* class with standard cross-entropy from the *trl* library.

In Section 4.6, we proposed adding new tokens to improve inference speed and decrease the tokenized text size. This means the embedding layer needs to be extended, which introduces two problems. First, the decision on how to initialize the new embedding weights, and second, how to further integrate it with QLoRA.

After extending the embedding layer, the new embeddings' weights need to be initialized. We decided to initialize the weights to the mean of the existing embeddings. The existing libraries support integration with QLoRA, but we noticed an incompatibility between library versions. It is also required to follow the proper order for applying the embedding layer, unfreezing specific model layers, and setting up QLoRA adapters. The QLoRA settings we used are rank = 16, alpha = 32, and dropout to 0.05.

Before training, the model is quantized to 4 bits, enabling it to be trained on consumer-grade hardware such as 24GB VRAM graphics cards. To further lighten the training, we use QLoRA. Here, one must be careful because the weights of the original layers are frozen.

This is not a problem, but remember, we also want to train the new embeddings. For that purpose, explicit unfreezing of the embedding layer and the model's head is required. While posing a challenge, new tokens reduced the number of tokens in one prompt-expected output pair by 37.5%. The experiments with the fine-tuned model are described in Chapter 6.

5.5 Algorithms calling LLM

A big part of experiments is based on calling LLM in algorithms repeatedly, namely refinement loop with HCRL or LVRL solution strategies (Section 4.8), evolutionary (Section 4.9), and coevolutionary algorithms (Sections 4.9 and 4.10). All the loop and evolution algorithms use *llm_utils.py*, which governs fine-tuned LLM initialization, batch inference, and also API LLM calling. The main part of all iterative and evolution-based algorithms starts in *evolution_loop.py* script.

Each algorithm, the refinement loop with hill climbing and the last valid solution, described in Section 4.8, is implemented as a separate function. The evolutionary algorithms have a separate class. *EvolutionLocal* class implements the evolution with fine-tuned LLM, and *Coevolution* class implements the coevolution with API models. The evolution and coevolution classes share methods and differ in the circuit generation.

We tested the layer inversion (Section 4.7) in single-call and loop experiments. It is implemented by attaching a PyTorch forward pre-hook to the MLP submodule of a selected transformer layer. This mechanism allows modifications of hidden states during inference without any further model modifications.

5.5.1 Coevolution

To properly steer the LLM, we constructed an algorithm based on coevolution. Evolution is part of the refinement loop framework. The parameters and source paths are defined using a YAML configuration file.

The evolution algorithm coevolves a population of prompt templates and circuits. The algorithm starts by initializing the circuit and the prompt population. The initial circuit population consists of a subset of EvoApproxLib approximate circuits. The algorithm loads and initiates the prompt population from a handwritten set of prompt templates. The algorithm is implemented as a single loop, where each generation performs the following steps in order: generate circuits, select circuit survivors, select prompt survivors, and generate new prompt templates. Each step has its method.

Circuit generation has three nested loops. The outer loop iterates over targets. In early experiments, we found that the algorithm performs better for a single target, so the option to define multiple targets and average the fitness function values will not be described further. Parameters of a non-existent circuit that the algorithm is supposed to discover constitute a target. The target/s are defined and loaded from the YAML configuration file. Based on the current target, the middle loop identifies circuits with specific properties, such as being closest to the target or having the lowest error, from the current circuit population. In the innermost loop, the algorithm inserts these values into the prompt template. The innermost loop iterates over the prompt templates population and, for each template, constructs a prompt by inserting data extracted in the middle loop. The middle loop runs the innermost loop with different random circuits. After the innermost loop prepares prompts, the middle loop calls LLM with all the prepared prompts. When the new circuits are generated, the middle loop calls circuit evaluation, and the corresponding circuit and prompt template

statistics are updated based on the circuit measurements. The configuration file defines the number of the middle loop repetitions by the parameter „repetition“, k in Section 4.10.

Custom functions extract circuit data from the current population, and prompt templates use selected data to construct prompts for the LLM. The processing functions are *__smallest_metric_value(metric)*, *__closest_to_target(target)* and *__random_circuit()*. Each method returns a structure containing circuit errors, area, and design. Method *__smallest_metric_value(metric)* returns a circuit with the smallest value of the selected metric. The method *__random_circuit()* returns a random circuit from the population, and the method *__closest_to_target(target)* returns the two closest circuits to a target specified by the parameter. The *__closest_to_target(target)* method compares values normalized by the target values; another option is to normalize values based on the current maximum values, which led to an imbalanced influence of each metric (for example, a small difference in area was more dominant than a large difference in error).

A new population of circuits is selected using the `RankAndCrowdingSurvival` function from the *pymoo* library. It uses a survival strategy based on the NSGA-II algorithm. The `RankAndCrowdingSurvival` also accepts multiple predefined crowding functions. We use Pruning Crowding Distance [16]. During early experiments, we also tried M-Nearest Neighbors [16] when we used three or more fitness functions. The new population contains the best of the combined pool of the previous population and the new circuits, which is called $(\mu + \lambda)$ selection. For circuit selection, fitnesses are MSE and area. Based on the evolution objective, WCE replaces the MSE in the fitnesses. MAE for multiplication by zero must be equal to zero, and this condition is set as a constraint. This constraint is necessary for use in the deep neural network accelerators [22].

A new population of prompt templates is selected using the same strategy as the circuit population. It uses different fitness functions, namely the average difference between the current and target MSEs, and the one minus the fraction of valid responses, defined by equation 4.3. WCE may replace the MSE in the fitness according to the evolution objective.

The last step of coevolution is creating new prompt templates. The surviving prompt templates undergo crossover and mutation, which are performed by an LLM using a randomly chosen handwritten prompt template (meta-prompt template). The algorithm adds the new prompt templates to the prompt template population for the next generation.

The same LLM is called with the conversation history after each generation to extract only the new circuit or the new prompt template. The algorithm terminates after a certain number of generations.

Chapter 6

Experiments

The proposed solutions and algorithms were implemented, and the results of the experiments are described in this chapter.

First, we focus on the model selection using few-shot prompting. Then we proceeded with few-shot prompting experiments, and determined the number of examples to use for the best results. After that, we moved on to experiments with a trained model, exploring the effects of different settings on the quality of LLM outputs.

6.1 Model comparison

In Section 4.3, we selected and described four LLM models. Now we present the experiment that compares them to determine which LLM type is most suitable for further investigation. The dataset for the measurement is EvoApprox8b; specifically, the 8-bit unsigned multipliers part is used. To test untrained LLMs, we decided to include two formatted circuits in the prompt.

6.1.1 First prompt

The first version of the prompt is in Listing 6.1. The example circuits were selected to have similar MSE values. That ensures a fair evaluation and properly guides the model.

```
You are a professional hardware designer. You specialize in approximate
circuits. Carefully follow the equation format in examples. Please, design a
new approximate multiplier with lower Mean Squared Error (MSE) and do not
increase the area of the final circuit compared to the following examples.
Example 1: MSE = <MSE1>, Area = <AREA1>
<CIRCUIT1>
Example 2: MSE = <MSE2>, Area = <AREA2>
<CIRCUIT2>
```

Listing 6.1: Prompt version 1

6.1.2 Improved prompt

Based on the LLM output analysis, the prompt was manually modified, and we will refer to this prompt as version 2. The improved prompt still uses 2 examples. We removed the ambiguity in defining how small the MSE and the area to generate should be, and we added a percent threshold to the prompt. The model is asked to generate only one design.

Also, it is explicitly stated that the design should be generated without any description. The role (professional hardware designer) is removed because it confused the model. An explicit hint to generate the new design is added at the end of the prompt. The limit on the number of generated tokens (max_new_tokens) is increased from 2048 to 4096. The improved prompt is in Listing 6.2.

```
Following equations describe approximate multiplier. Carefully follow the
equation format in examples and generate exactly one new design. Use exactly
the same variable names and syntax as in the examples. Do not include
explanations, output only the equations. Please, design a new approximate
multiplier with area within +-10% size of the examples and as low Mean
Squared Error (MSE) as possible compared to the following examples.
Example 1:
MSE = <MSE1>, Area = <AREA1>
<CIRCUIT1>
Example 2:
MSE = <MSE2>, Area = <AREA2>
<CIRCUIT2>
New design:
```

Listing 6.2: Prompt version 2

6.1.3 Observations

To evaluate each model's performance, the prompt-target parameters of the generated circuits are compared with those of the circuits in the prompt. Each model was given 255 prompts from each prompt template version. The computed ratios are shown in Figure 6.1

The first prompt version resulted in 16 acceptable circuits. By acceptable, it is meant that the MSE and estimated area (cost) are smaller or within a 10% tolerance of the minimums of their parents' parameter values. For the prompt version 2, this number increased to 53 circuits. These numbers are for all models combined, meaning 1020 inferences per prompt template. The successful circuits were generated primarily by the Qwen2.5-Coder-7B-Instruct model. Overall, syntactically correct results were generated mostly by Qwen2.5-Coder-7B-Instruct and Qwen2.5-Math-7B. Falcon3-7B-Instruct appears in the ratio graph, but it is more of an anomaly, because most of its outputs were syntactically incorrect. More Figures with graphs from this experiment are in appendix A.1.

The reasoning model (OpenReasoning-Nemotron-7B) keeps thinking, but does not generate a valid, comprehensive block of circuit code. By the thinking, it is meant that it describes the circuit modifications. It does not produce the requested design.

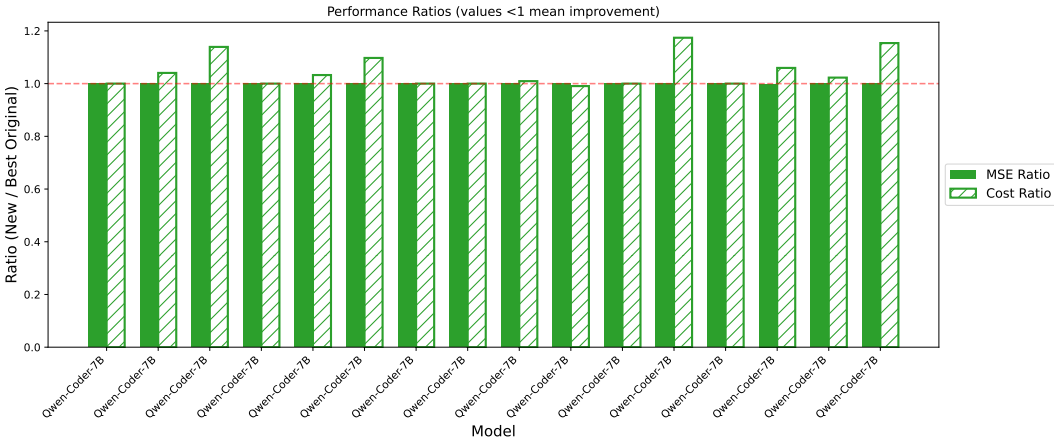
The general model (Falcon3-7B-Instruct) sometimes generates a description of what it would change or do, not the actual circuit code. However, it often generates valid code with some mistakes that cause problems in further use.

Math model (Qwen2.5-Math-7B) starts generating well, but at the end often forgets to add output assignment or generates repeating patterns (the examples are shortened)

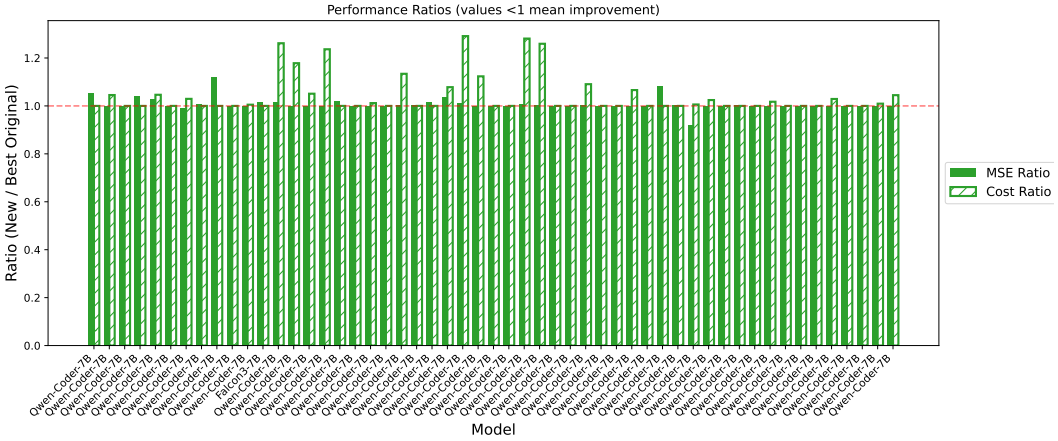
```
N7
N7
...
```

or

```
N20 = N21 = ... = N101 =102 = N103 =104 =105 ... =33
```



(a) Prompt 1: MSE and cost ratios



(b) Prompt 2: MSE and cost ratios

Figure 6.1: Results of few-shot prompting. The prompt contains two example circuits; the target is to minimize MSE while preserving area (cost). Columns express ratios between the minimal MSE from the given two example circuits and the MSE of the generated circuit, similarly for cost. Part (a) shows the results of prompting with the prompt template version 1, and Part (b) shows version 2.

The code model (Qwen2.5-Coder-7B-Instruct) follows the prompt most effectively, yielding the highest number of valid circuit designs. In fact, most of the valid circuits in this experiment were produced by this model.

6.2 Impact of few-shot example count on circuit design

This experiment aims to measure the number of example circuits in the prompt that yield the best new designs. Prompts in this experiment use a modified template from the previous experiment; the main difference is the addition of gate sizes, see Listing 6.3.

```
Following equations describe approximate multiplier. Carefully follow the
equation format in the examples and generate exactly one new design. Use
exactly the same variable names and syntax as in the examples. Do not include
explanations, output only the equations. The sizes of gates in square
micrometers are INV(~) = 1.40, AND(&) = 2.34, OR(|) = 2.34, XOR(^) = 4.69.
Please, design a new approximate multiplier with area within +-10% size of
the smallest example and as low Mean Squared Error (MSE) as possible compared
to the following examples.
Example 1: MSE = <MSE1>, Area = <AREA1>
<CIRCUIT1>
...
Example 4: MSE = <MSE4>, Area = <AREA4>
<CIRCUIT2>
New design:
```

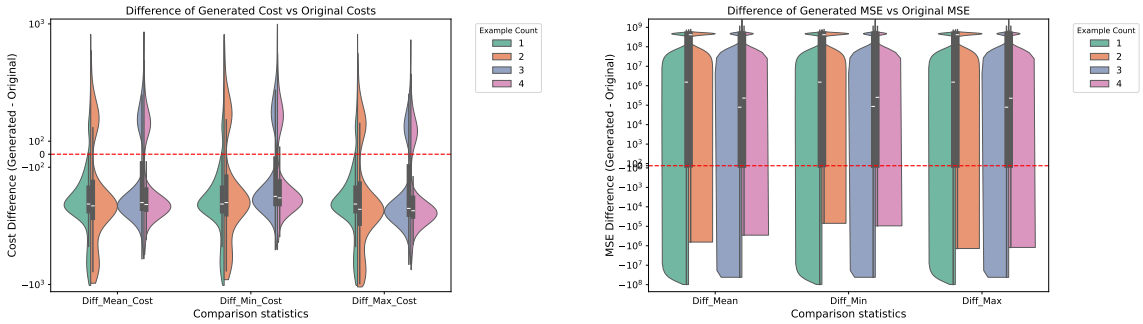
Listing 6.3: Prompt for few-shot experiments

Prompts containing one to four circuit examples were tested in this experiment. To fairly compare the impact of the example counts, 1,000 prompts are prepared for each count. To execute the experiment, at least 4,000 circuits are required to create the prompts. Therefore, these experiments use the 8-bit unsigned multipliers from the private EvoApproxLib dataset, which offers enough data. The circuits are sorted by MSE, and only the neighboring circuits whose areas differ by less than 25% are inserted into a single prompt.

6.2.1 Observation

The result of the experiment is that the one and three examples in the prompt yield the best MSE on average while keeping the area in limits, as Figure 6.2 shows. Besides the visualized data, there is also a slight increase in inference time with more circuit examples in a prompt, which needs to be considered and favors lower example counts. Besides the graphs, we also calculated the number of generated circuits that satisfy the prompt's constraints on MSE improvement and area retention. The results are in Tables 6.1 and 6.2. The tables also show another experiment, where the only difference is renaming the original nodes to start from N0, and each number follows the previous. This comparison is intended to investigate the impact of numbering on the generated circuit designs. The data shows a 4.0% decrease in syntactically correct generated circuits and around 0.725% decrease in prompt-target satisfaction. Fisher's exact test revealed this difference was not statistically significant, showing a trend toward significance with $p = 0.056$, and the original naming showed slightly higher odds of generating improved circuits. This is caused by the absolute success rate being quite low, around 3%, which makes the differences less significant. Also, the circuits were generated using an LLM set to the default temperature of 1.0, which intro-

duces randomness and influences the results. The comparison aims to determine whether sequential numbering imposes a significant drawback and whether it can be used to format the training dataset in the next step.



(a) Cost differences

(b) MSE differences

Figure 6.2: Measured differences between a) cost and b) MSE of generated circuit designs and the minimal, average, and maximal original circuit parameters.

Table 6.1: Impact of node naming scheme on circuit generation quality with varying example counts.

	Number of examples in a prompt				
Naming Scheme	**1**	**2**	**3**	**4**	**Total**
Original node names	49	12	25	17	103
Sequential (N0, N1, ...)	46	8	10	9	73

Table 6.2: Impact of node naming scheme on circuit generation quality. Displaying the produced circuit counts of three quality categories.

Naming Scheme	**Better than original**	**Prompt satisfying**	**Syntactically correct**
Original node names	103	103	3490
Sequential (N0, N1, ...)	73	74	3330

6.3 Fine-tuned model experiments

This section evaluates and analyzes circuits generated by Qwen2.5-Coder-7B-Instruct, fine-tuned for 1 epoch using a dataset prepared with *Train_generate_template*. The experiments in this section compare multiple LLM settings. The model was provided with parameter descriptions of circuits from a test subset of the private EvoApproxLib dataset. Each setting was tested on all 250 circuits of the test dataset.

6.3.1 Observation

From the global view, only one circuit dominated the circuits created by the CGP method. This circuit is generated with a temperature of 0.4. The other LLM-generated circuits in the graph are just blank codes, which cause the generated circuits to have zero area and thus appear in the Pareto front.

Even without achieving the CGP quality, changing the temperature and using the new inversion method seem beneficial. In the graphs shown in Figure 6.3, zero temperature is displayed separately to represent the default LLM output.

Changes in temperature lead to the finding of circuits with better parameters. With a lower MSE, the temperature choice does not change the result much. With a higher MSE, temperature 0.0 is stable, and around MSE 10^8, temperatures 0.4 and 0.6 are better options than 0.0, effectively generating circuits with better area and MSE.

Inversion of layers 23 and 24 is the most promising strategy for generating high-quality circuits, but inversion at layers 5, 8, and 9 also produced relatively good circuits. The inversion of layer 24 also exhibits improved prompt following, where 38 out of 250 generated circuits are within prompt boundaries, and 232 are syntactically valid, effectively the best ratio among all settings, as shown in Figure 6.4.

The rest of the results from this experiment are included in appendix A.2.

6.4 Bit-width generalization of circuit generation

We conducted experiments to test whether a fine-tuned model can generate circuits of different bitwidths. We particularly focused on generating 7-bit and 11-bit circuits using EvoApproxLib$^{\text{LITE}}$. We created the prompts for 18 7-bit and 11 11-bit circuits. The model used in these experiments is Qwen2.5-Coder-7B-Instruct, fine-tuned for 1 epoch on *Train_generate_template*.

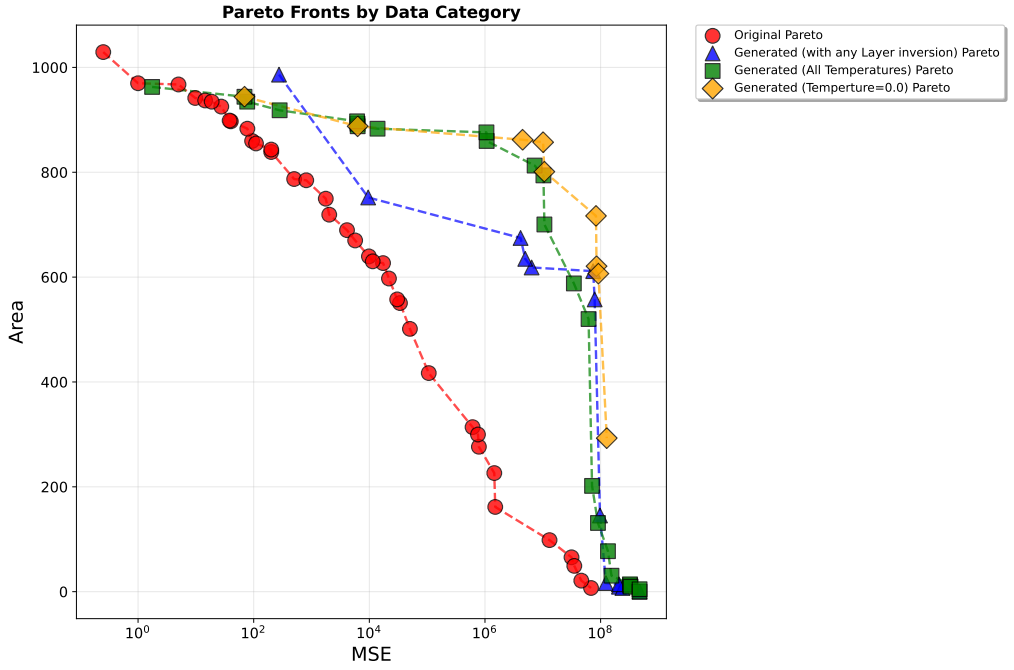
Three versions of prompts were tested:

1. The original prompt used for training with changing part *'generate the corresponding 8-bit unsigned'* to *'generate the corresponding bits-bit unsigned'* where *bits* placeholder is substituted with 7 respectively 11.

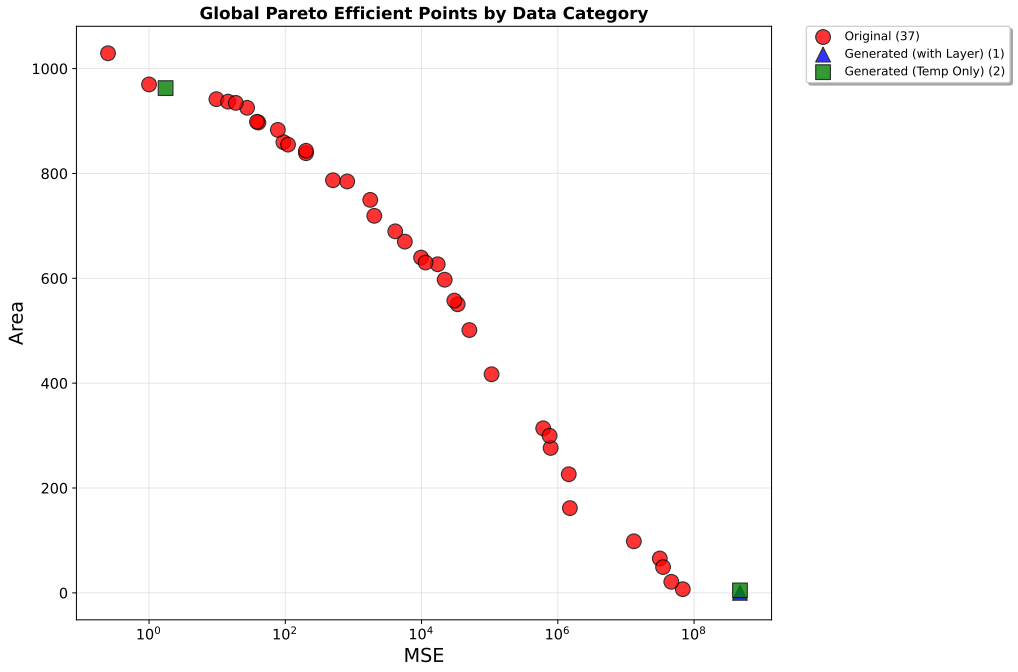
2. Based on results from version 1, we explicitly instruct the LLM to include all required inputs for the 11-bit circuit, listing them in order from INA0/INB0. We also command the LLM not to include excessive inputs from 8-bit circuits in 7-bit designs.

3. Similar to version 2, but contains only the additional inputs IN(A/B)8/9/10. The adjusted part for 11-bit circuits is *'the circuit takes 11-bit inputs, so you must also use inputs INA8, INA9, INA10, INB8, INB9, and INB10:'*. For 7-bit circuits, the prompt lists only the redundant inputs INA7/INB7, instructing not to use them. The adjust part of the command is *'the circuit takes 7-bit inputs, so you must not use inputs INA7 and INB7:'*.

The generated circuit designs were automatically checked to ensure they do not contain more or fewer inputs than specified. The fine-tuned LLM did not adhere to the 7-bit constraint and generated designs with inputs for an 8-bit circuit. Two 11-bit circuits passed the test. We manually inspected them and found that the designs are invalid. The LLM did not include an operator between the input 10 and the other operands, making the designs



(a)



(b)

Figure 6.3: Pareto dominant circuits. Comparing circuits generated by LLM with different settings to the ground truth from the EvoApproxLib dataset. On the top (a), there are separated Pareto fronts for each type of setting, and on the bottom (b), there is a global Pareto front.

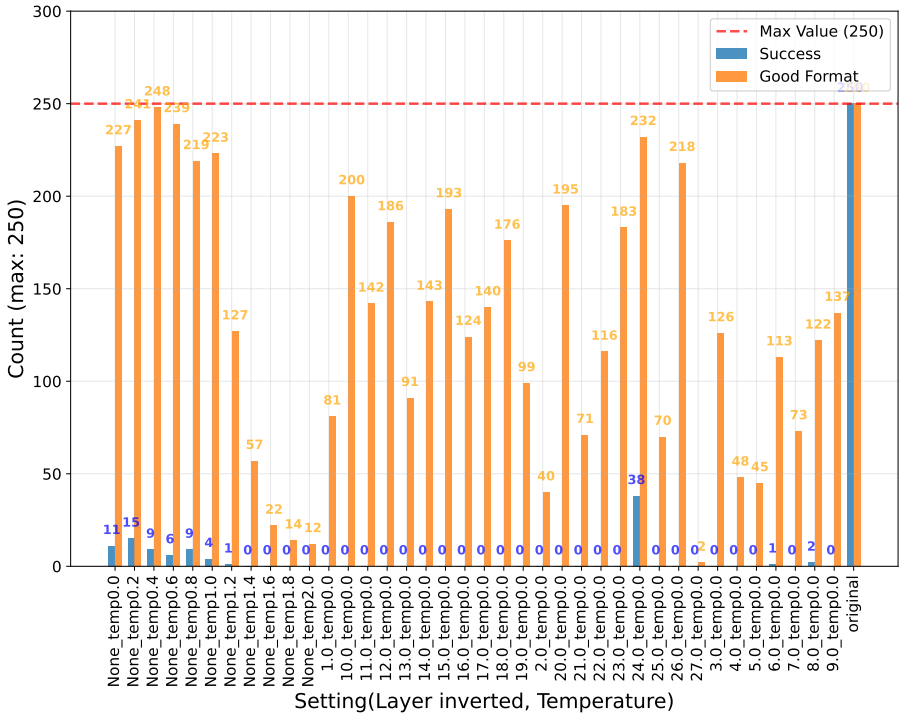


Figure 6.4: Syntactical and semantical correctness of the circuit designs generated with LLM with various settings of temperature and layer inversion.

invalid. We present the error in the first design: *N0 = INB10 INB0 INA0*, and in the second design: *N0 = INB10 (N0 INB0)* .

The conclusion is that the model trained for 8-bit circuits cannot be used to generate designs of different bitwidths. While there are some signs of plasticity (the LLM includes the INA10 input), the LLM can not understand how to logically and meaningfully incorporate it into a circuit.

6.5 LLM in a loop

Now, we describe the experiments where the LLM is part of the refinement loop as introduced in Section 4.8. The experiments utilize a fine-tuned LLM. We fine-tuned the Qwen2.5-Coder-7B-Instruct model on a mix of prompts generated from *Train_generate_template* and *Train_improvement_with_target_template*. The order of parameters in the examples is shuffled, which encourages the model to learn associations between the parameters and corresponding circuit designs. We fine-tune it for 8 epochs. Figure 6.5 shows the fine-tuning progress. There is a gap between the fine-tuning and evaluation loss. Since the evaluation is performed on unseen data, the gap reflects generalization error. The evaluation loss remains consistent over the epochs without significant widening. This indicates model generalization without overfitting.

We extracted target circuits and their parameters from the test split of the prepared dataset, before the experiments. The algorithm's goal is to generate the target circuit. The experiments start by prompting the fine-tuned LLM to generate a circuit with the target parameters from scratch. The LLM is asked to generate a circuit with the target

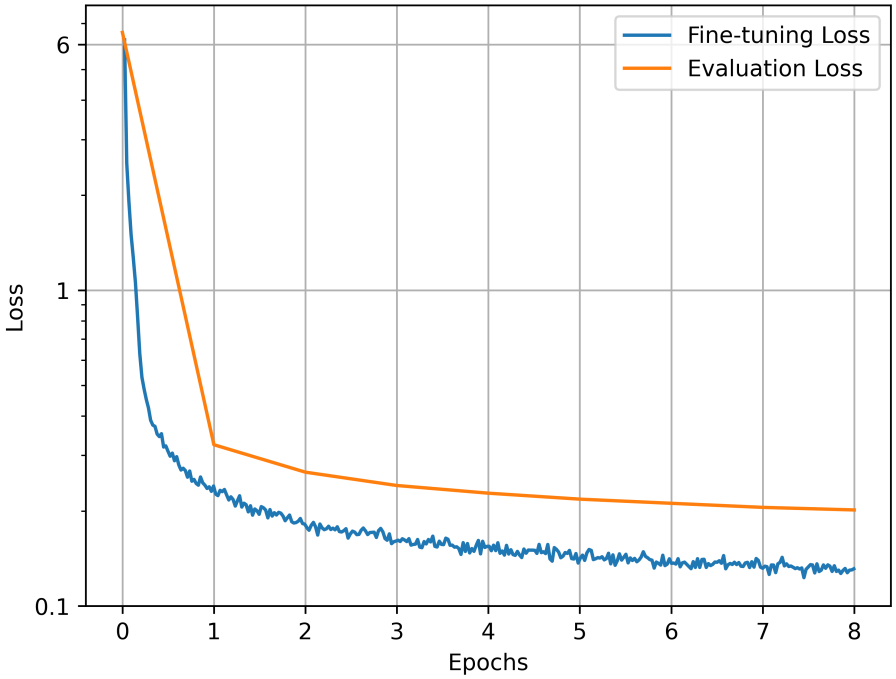


Figure 6.5: Progress of Qwen2.5-Coder-7B-Instruct fine-tuning for 8 epochs.

parameters and is also given an example circuit, as in the previous experiment. We ran experiments with two algorithms: Hill Climbing Refinement Loop (HCRL) in Algorithm 2 and Last Valid Refinement Loop (LVRL) Algorithm 1, which differ only in the choice of the example circuit from previous iterations. HCRL takes the best circuit so far. The best circuit dominates the former circuit. The LVRL algorithm uses the circuit from the previous iteration as an example circuit for the current iteration. We ran the experiments for 500 iterations. The experiments were set to use the same initial circuits and targets for HCRL and for LVRL.

6.5.1 Observations

The setup enables us to compare the two algorithms directly. We can not clearly decide which algorithm performed better than the other based on the plotted data in Figure 6.6. Each algorithm produces circuits that dominate the other algorithm's circuits in different regions. We had to compute the hypervolume of each algorithm's Pareto front to compare them properly.

A reference point is required to compute the hypervolume. The reference point is computed as a data point with the worst MSE-area increased by 1%. First, we compare the algorithms' Pareto fronts separately. The HCRL generated 72 Pareto front points, while the LVRL algorithm generated 63. However, the hypervolume comparison shows the HCRL has a 0.97% smaller hypervolume. We also combined the generated data from both algorithms and computed a mutual Pareto front. The hypervolume was compared by splitting the points of the mutual Pareto front into groups based on the algorithm that generated them. It shows that the hypervolume of the HCRL is 2.74% smaller than the hypervolume of the LVRL algorithm, when considering the mutual Pareto front points.

The HCRL, however, generated more Pareto front points, 58, while the LVRL generated 46. The numbers favor the LVRL algorithm over the HCRL.

It is also worth noting that both algorithms produced designs which are Pareto dominant over targets or at least close to the targets in the region with MSE in interval $< 0; 400)$.

The goal of this experiment was to rediscover existing circuits from the test dataset, the ones LLM was not trained on, nor were hyperparameters tuned on them. Despite that, the algorithms generate a few non-dominated circuits. We further investigated them and found out that 12 are degenerated circuits with area = 0, and 3 are valid designs. The algorithms did not require MAE for multiplication by zero to be zero, so we had to test this property separately. All three satisfy MAE_zero = 0. The LVRL produced two Pareto-dominant circuit designs across all runs. The generated circuits have the following parameters: (area = 4.7 μm^2, MSE = 6.51×10^7) and (area=976.8 μm^2, MSE=0.25). The HCRL algorithm produced a circuit with an area of 941.7 μm^2 and an MSE of 6.5.

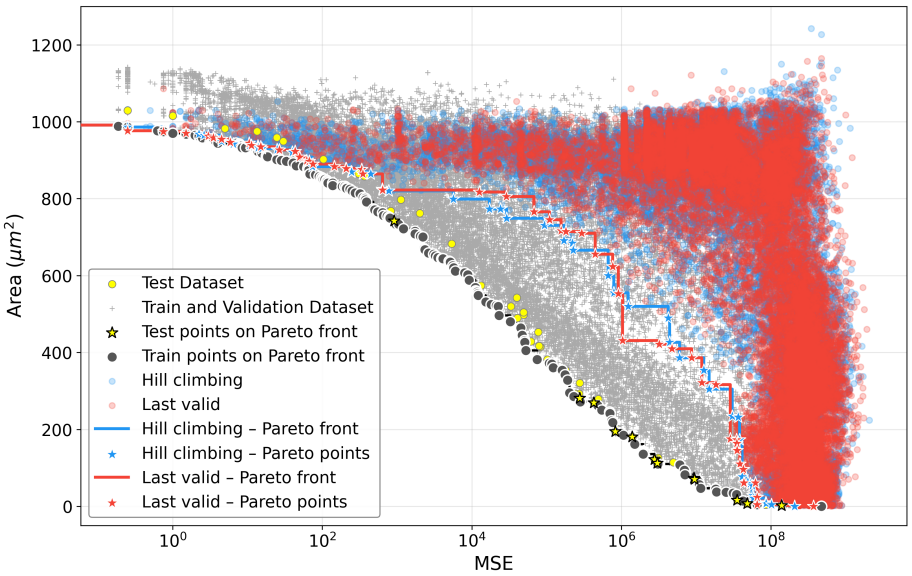


Figure 6.6: Comparison of 500 iterations with fine-tuned LLM.

6.5.2 Extended experiments

To further enhance the experiments, we doubled the number of iterations and added layer inversion (described in Section 4.7). We specifically targeted the region with the largest gap between generated designs and the targets.

Additional iterations produced new non-dominated points, as seen in Figure 6.7. The newly produced designs slightly improve the Pareto front. However, they do not significantly overcome the gap between generated and target designs.

In previous experiments, we observed that the layer inversion can enhance LLM inference results. We also identified the layers where the layer inversion application has the greatest positive effect. Now we run the loop experiments for 500 iterations with layer inversion at layer 8 (Figure 6.8) and layer 24 (Figure 6.9) separately. The inversion at layer 8 caused algorithms to collapse into a low area and a high error region. The inversion at layer 24 helped the algorithms find new non-dominated trade-offs (around MSE 10^3, 10^4 and 10^7). When considering the MSE interval $(10^4, 10^7)$ and the inversion of layer 24, the Pareto fronts

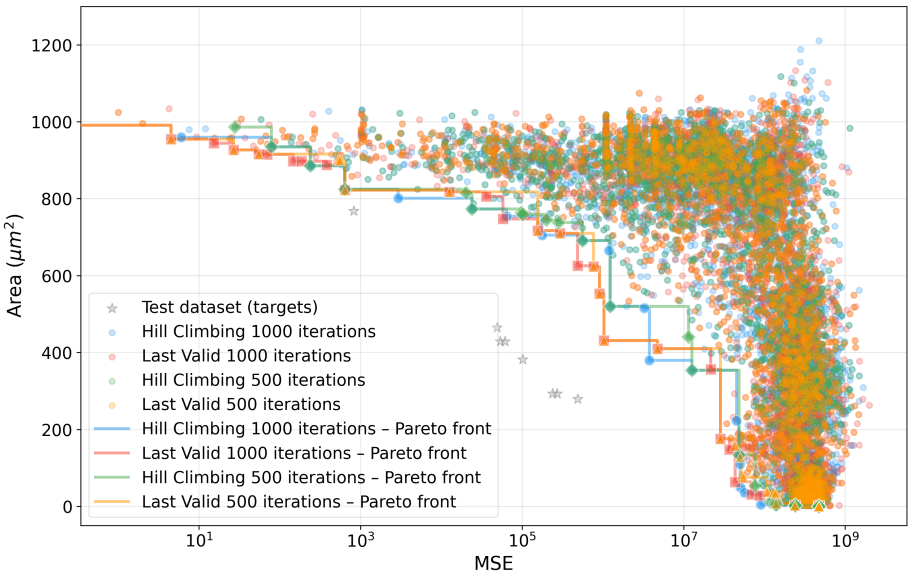


Figure 6.7: Comparison of 500 and 1000 iterations. Refinement loop algorithms with fine-tuned LLM.

are worse (pushed to the upper right corner). The benefits of layer inversion observed in previous experiments are dampened or completely diminished in loop experiments. The main purpose of the inversion technique is to increase the exploration. We hypothesize that high temperature and the iterative nature of loop refinement provide sufficient exploration, rendering layer inversion redundant.

6.6 Evolution with fine-tuned model experiments

This experiment integrates a fine-tuned model into the evolutionary loop. This experiment aims to improve the results of previous experiments. We fine-tuned the LLM solely to improve one metric, while keeping the others almost unchanged. Then we inserted this model into an evolutionary algorithm.

The algorithm starts with existing circuits sampled from EvoApproxLib. The model is given a circuit and asked to improve the error (WCE or MSE) and area in separate prompts. After each iteration, only the best circuits are kept for the next population using `RankAndCrowdingSurvival` function from *pymoo*. The algorithm terminates after fixed number of iterations.

We fine-tuned the model only on improvement prompts (*Train_metric_improvement_template*). To prepare the prompts, we used the function *find_neighbors*, which selects circuit pairs, where the second circuit improves one of the metrics WCE, MSE, MAE, or area compared to the first circuit in the pair. Thresholds regulate the difference between the circuits in the dataset. The first circuit is given to the LLM; it tries to find a better circuit, and the second circuit is the expected answer. We fine-tuned the Qwen2.5-Coder-7B-Instruct model on this dataset for 8 epochs.

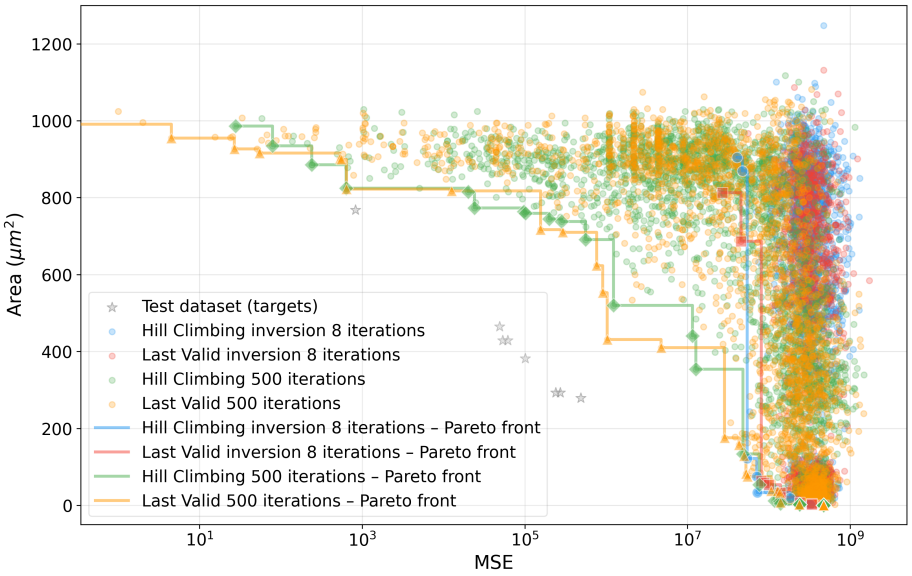


Figure 6.8: Layer inversion at layer 8. Refinement loop algorithms with fine-tuned LLM.

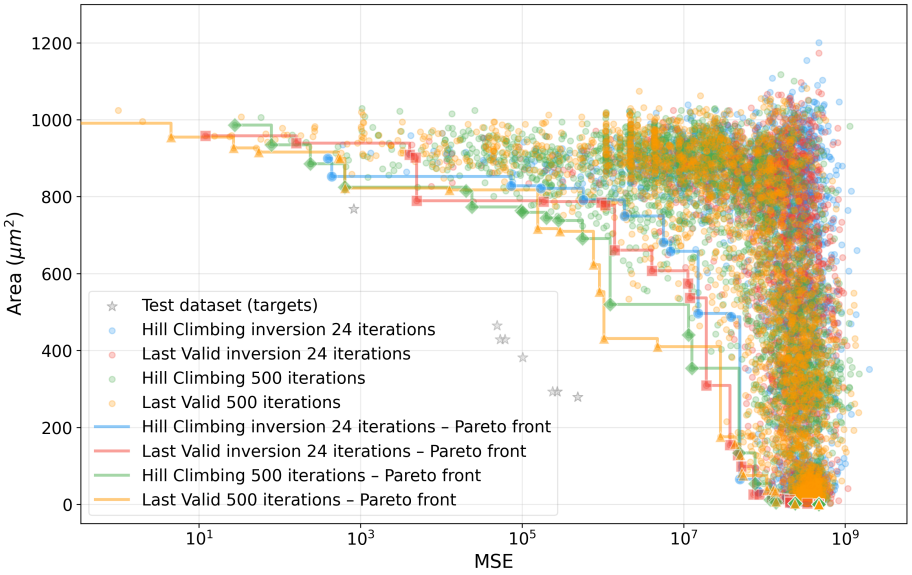


Figure 6.9: Layer inversion at layer 24. Refinement loop algorithms with fine-tuned LLM.

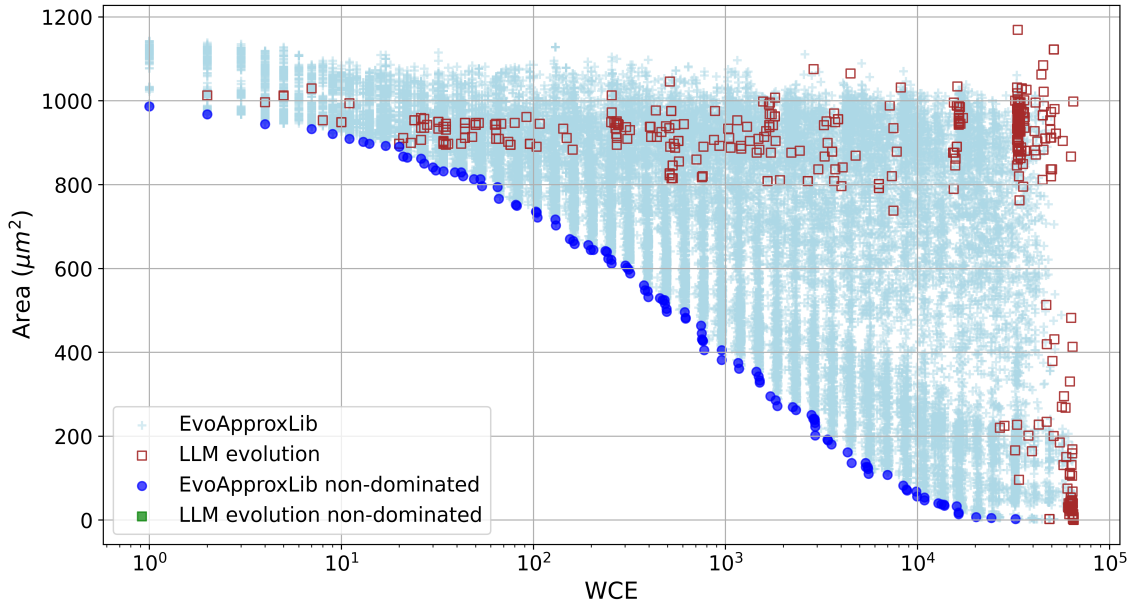


Figure 6.10: The *WCE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed Evolution with fine-tuned model (brown squares) or taken from EvoApproxLib (blue dots or light blue crosses).

6.6.1 Observations

The algorithm tends to generate circuits at the edge cases, where either the error or the area is low. It is similar to the previous experiment observation (Section 6.5). The results of evolution for WCE-area trade-offs are shown in Figure 6.10, and for MSE-area trade-offs are in Figure 6.11. We ran the experiments for 10 generations, then increased the number of generations to 125 (to roughly match the number of LLM calls in the subsequent experiments). However, the algorithm quickly converges and does not recover afterward. Despite that, the algorithm manages to generate near-Pareto-front circuits with WCE around 2000. The limitations of this and previous approaches motivated the creation of coevolution of circuits and prompt templates.

6.7 Evolution with API models experiments

We performed experiments with coevolution as described in Section 5.5.1. First, we determined the parameters by trial and error using shorter runs. The statistical evaluation of all the parameters is intractable. We ran experiments with prompt crossover and a mutation probability of 0.9, a repetition parameter of 3, for 27 generations. The algorithm starts with loading 10 source circuits; they are randomly sampled from a prepared pool of circuits from EvoApproxLib. The prepared pool contains circuits sampled from a broader error/area region around the original target. The circuits evolve towards a user-defined target. The target is defined by the parameters of an artificial circuit, derived from a circuit in EvoApproxLib with lowered parameter values. We determined the target from one of the Pareto-dominant circuits (original target) by reducing its parameter values to 90, 85, or 70 percent.

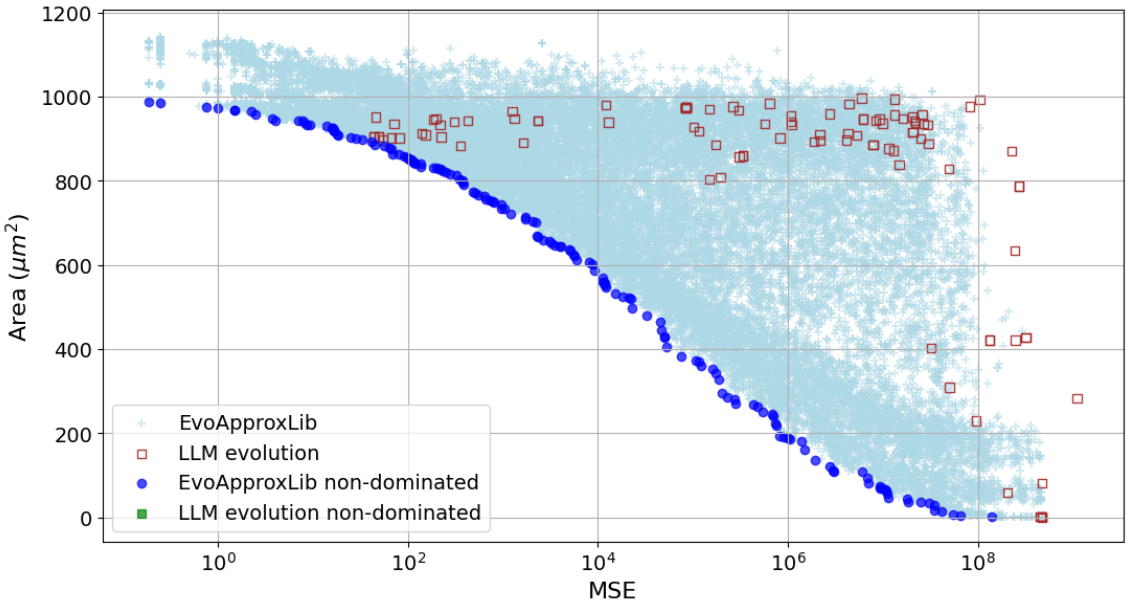


Figure 6.11: The *MSE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed Evolution with fine-tuned model (brown squares) or taken from EvoApproxLib (blue dots or light blue crosses).

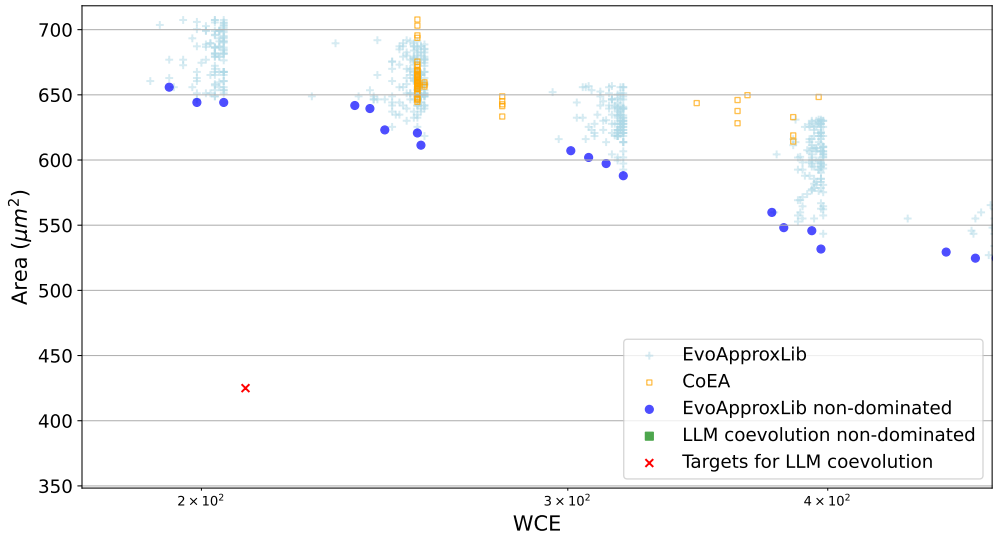
For circuit design, we used Qwen3-Coder-480B, which is a mixture-of-experts coding model with 480B total parameters (approx. 35B active per inference), optimized for long-context (up to 256k tokens) code generation, reasoning, and agent-style software engineering tasks. For prompt templates, we used GPT-OSS-120B, which is a 120B-parameter open-weight LLM optimized for general reasoning and coding.

6.7.1 Observations

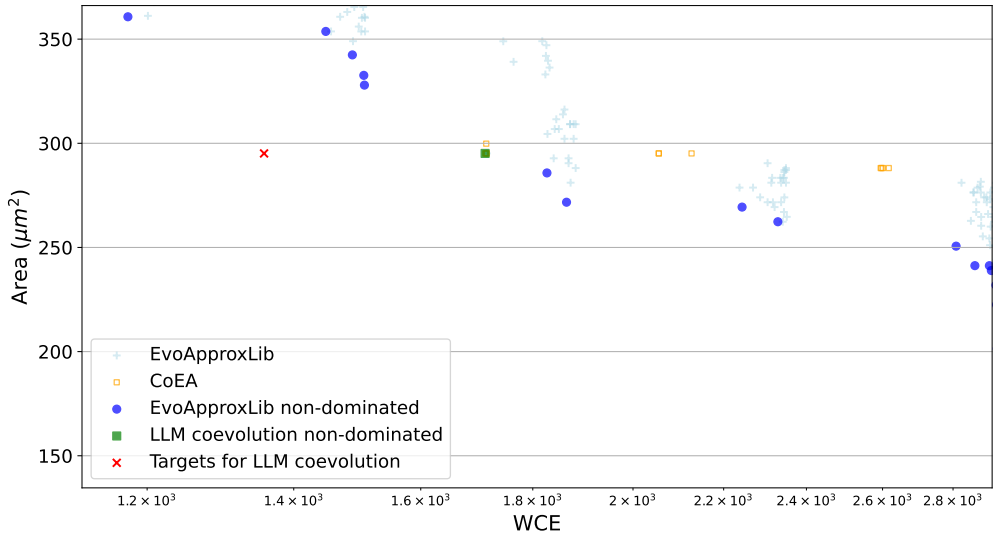
When considering area estimates, the algorithm successfully produced Pareto-dominant circuit designs in both MSE vs. area and WCE vs. area optimizations. The synthesis was performed by Synopsys Design Compiler for a 45 nm process. After the synthesis, the circuit's actual area is larger than the estimate. So the generated circuits are no longer better than those on the Pareto front but close to them in some regions.

We compared different initializations. When the initial circuits are of worse quality, sampled from a region maximally 150% worse in both objectives from the Pareto front, the algorithm gets close to the Pareto front (top Figure 6.12). The algorithm initialized with circuits showing parameters 3-8% worse than those occupying the Pareto front (bottom Figure 6.12) can produce circuits with similar parameters as those situated on the Pareto front. To reach Pareto front in fewer iterations in longer runs, we used circuits from a broader space around the Pareto to create the initial pools per target, from which the algorithm samples an initial population randomly.

Observed errors in generation. We observed cases where the new circuit was in Verilog, even though we did not intend to ask the LLM to generate Verilog code. Sometimes LLM adds a semicolon after each line or equation, which is a sign of misinterpreting the code for C, Java, or a similar language. These cases are rare, but they show that the LLM uses knowledge from pretraining on various kinds of code data.



(a) significantly worse initial circuits



(b) 3-8% worse than Pareto front initial circuits

Figure 6.12: CoEA with different initialization.

6.7.2 Analysis of CoEA

We consider the number of calls for the bigger LLM (Qwen3-Coder-480B) per run (LLMpR) as a computational cost indicator. With LLMpR = 450, we compare the quality of resulting Pareto fronts for 10 independent runs of three search algorithms targeting an 8-bit approximate multiplier with $T_E(WCE) = 21387$ and $T_A = 453$ μm^2. These target values are challenging as seen in Figure 6.13. They represent a hypothetical multiplier. The algorithms considered in the comparison are:

A hill-climbing algorithm (HC) starts with a circuit selected from EvoApproxLib, which is 1.2×–1.5× worse in both criteria than the target. It applies an LLM-driven mutation based on a single mutation template. The next parent is determined as the circuit with the smallest differences in both error and area from the target reached during the run. There is no co-evolution of templates. In each run, g = LLMpR iterations are executed.

The proposed coevolutionary algorithm is used in two settings. LLM is called once ($k = 1$) per template, and $g = 15$ generations are produced; we refer to this setting as CoEA-1-15. The proposed coevolutionary algorithm, with the LLM called 3 times ($k = 3$) per template and $g = 5$ generations produced. As $k > 1$, g has to be proportionally reduced to keep LLMpR constant. This setting is named CoEA-3-5.

Figure 6.13 compares the final Pareto fronts from all runs of the algorithms considered. HC produces the least competitive results and tends to converge to a few almost identical solutions. The use of multiple LLM runs per template is a key feature of the proposed CoEA. With $k = 3$, CoEA produced the highest number of unique solutions on the Pareto front; see the orange squares in Figure 6.13. Under this setup, the search is rather exploratory as many solutions "far" from the specification are reached. Figure 6.14 analyzes the hypervolume indicators computed for each final Pareto front. A hypervolume is the dominated area in the objective space with respect to a reference point defined as the worst observed values increased by 1%. CoEAs clearly outperform the HC algorithm.

The progress of evolution is visualized in Figure 6.15, where the hypervolume indicator is represented using a box plot of the best solutions (in 10 runs) reached in a given generation. It is important to note that CoEA-3-5 is executed for only 5 generations to ensure that both CoEA-3-5 and CoEA-1-15 use the same number of LLM calls. Based on visual comparison of Figure 6.13, 6.14, and 6.15, CoEA-3-5 seems to perform better than CoEA-1-15. However, a detailed statistical evaluation reported in Table 6.3 reveals that under the Mann–Whitney U test ($\alpha = 0.05$), no statistically significant differences are observed between CoEA-3-5 and CoEA-1-15, indicating comparable performance. Interestingly, CoEA-3-5 outperforms CoEA-1-15 if CoEA-1-15 uses slightly fewer resources (14 generations only). The average execution time of CoEA-3-5 is slightly shorter than that of CoEA-1-15 (207 min. vs 223 min).

We measured the time of a single call to the LLM across multiple runs. The time between calling Qwen3-Coder-480B with existing circuit examples and receiving a response with a newly generated design is about 10.0 seconds on average. The smaller model (GPT-OSS-120B) responds to a template-modification prompt on average in approximately 5.3 seconds.

By contrast, evaluating the error of a single candidate circuit takes only about 0.5 seconds on average, indicating that LLM inference dominates the overall runtime. Since the models were accessed via the shared computational infrastructure e-infra.cz, their inference times were beyond our control; this factor also influenced the time measurement, and the reported timings should therefore be considered indicative only.

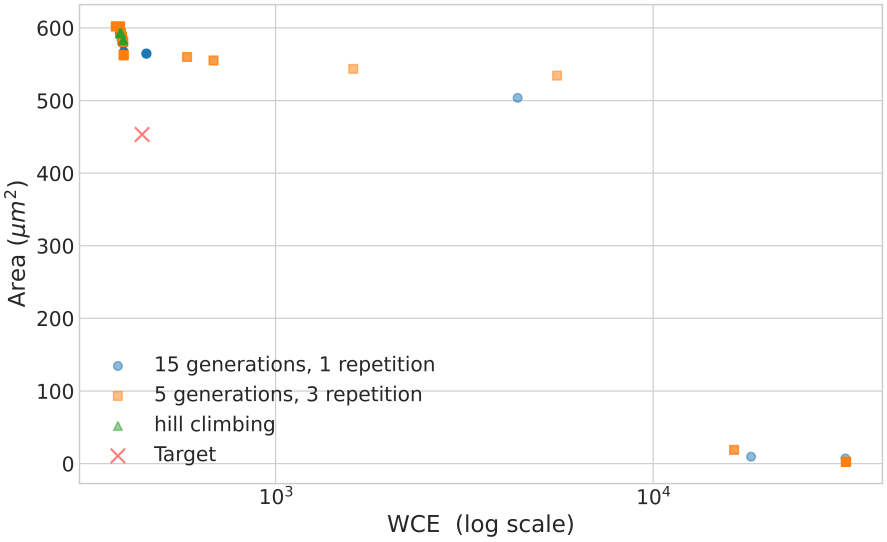


Figure 6.13: Final Pareto fronts obtained from 10 runs of HC, CoEA-3-5, and CoEA-1-15. The same initial population of circuits was used in the i-th run of CoEA algorithms. The target specification is depicted with a red cross.

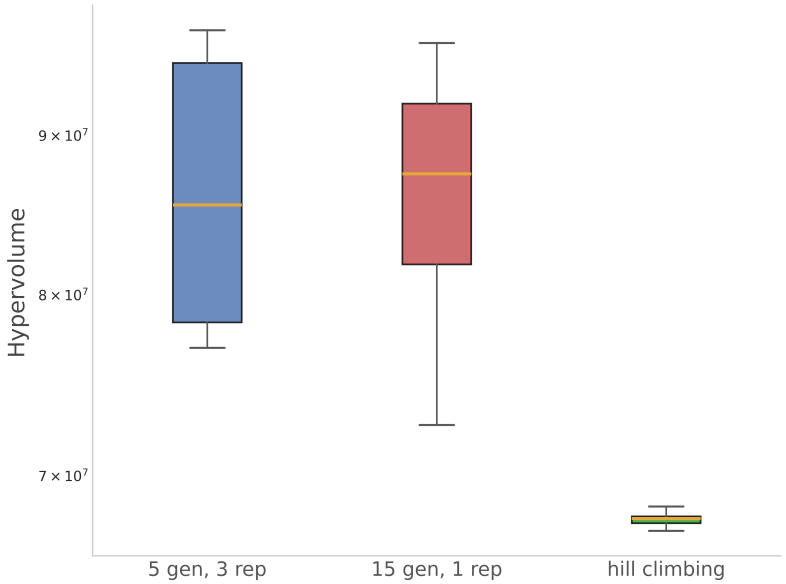


Figure 6.14: The hypervolume indicator computed from the final Pareto front constructed from 10 independent runs of HC, CoEA-3-5, and CoEA-1-15.

Table 6.3: Results of the Mann-Whitney U test for hypervolume comparison across experimental groups.

Comparison	Groups	U	p-value	Significant
Overall hypervolume (Fig. 6.14)				
CoEA-3-5 vs CoEA-1-15	overall	52.00	0.9097	No
CoEA-3-5 vs hill climbing	overall	100.00	0.0002	Yes***
CoEA-1-15 vs hill climbing	overall	100.00	0.0002	Yes***
Final generations comparison (Fig. 6.15)				
CoEA-3-5 (gen 5) vs CoEA-1-15 (gen 15)	per-gen	75.00	0.0638	No
CoEA-3-5 (gen 5) vs CoEA-1-15 (gen 14)	per-gen	80.00	0.0256	Yes*

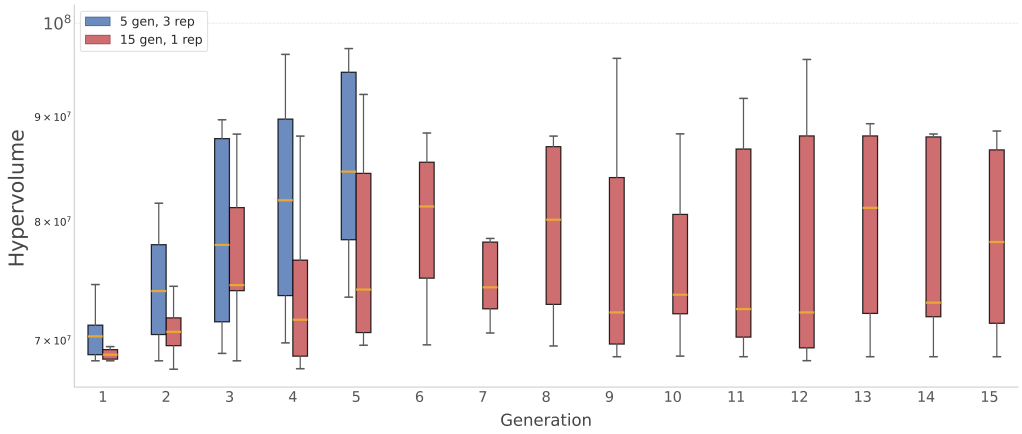


Figure 6.15: The progress of evolution of CoEA-3-5 and CoEA-1-15, where the hypervolume indicator is represented as a box plot of the best solutions (in 10 runs) reached in a given generation.

6.7.3 Long runs

Based on the results reported in the previous section, we used CoEA with parameters $k = 3$ and $g = 27$ to design approximate 8-bit multipliers for 25 target WCE-area pairs and for 14 target MSE-area pairs. Red crosses represent the targets in Figure 6.16. A single run, executed for each target pair, produced approximate multipliers that occupy a small local Pareto front. All global Pareto optimal designs produced by CoEA are depicted with green squares, and the Pareto optimal designs filtered from EvoApproxLib according to the WCE and area are shown as blue dots in Figure 6.16.

In 72 cases, approximate multipliers produced by CoEA show new trade-offs between WCE and area that are not present in EvoApproxLib; 24 of them are on the global Pareto front of all 8-bit multipliers. The light blue plus symbols and the open orange squares in Figure 6.16 denote multipliers that are dominated by other designs.

Experiment with MSE as target error metric (Figure 6.17) shows that CoEA can deliver competitive approximate multipliers. In this case, 65 implementations exhibit trade-offs not seen in EvoApproxLib, and 23 of them lie on the global Pareto front.

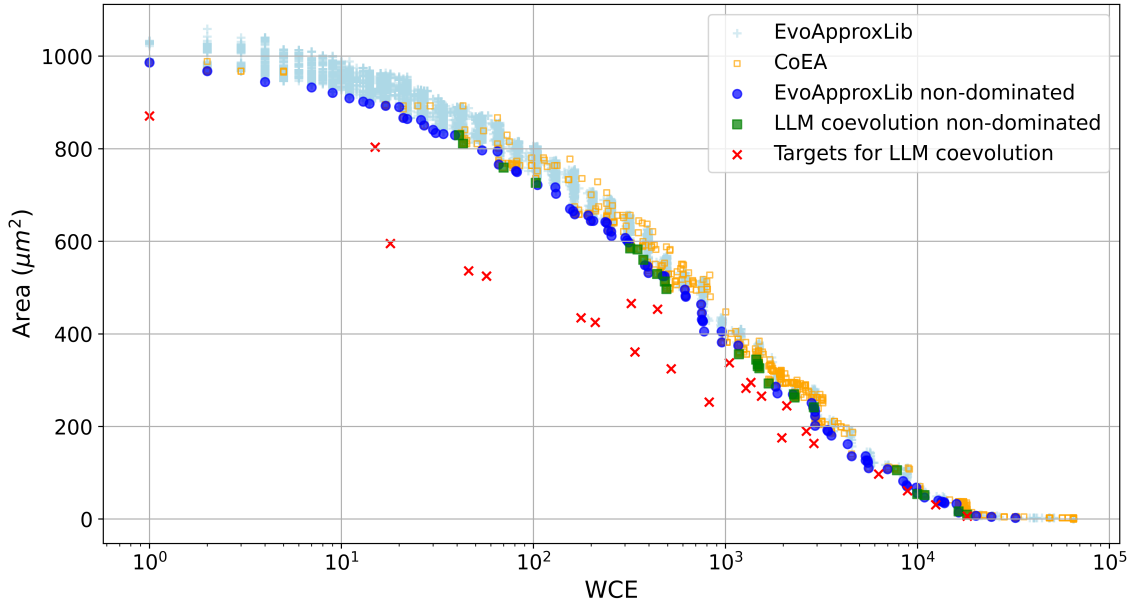


Figure 6.16: The *WCE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed CoEA or taken from EvoApproxLib. The united global Pareto front contains non-dominated solutions generated by CoEA (green squares) and taken from EvoApproxLib (blue dots). The remaining points are dominated. The target specifications provided to CoEA are depicted with red crosses.

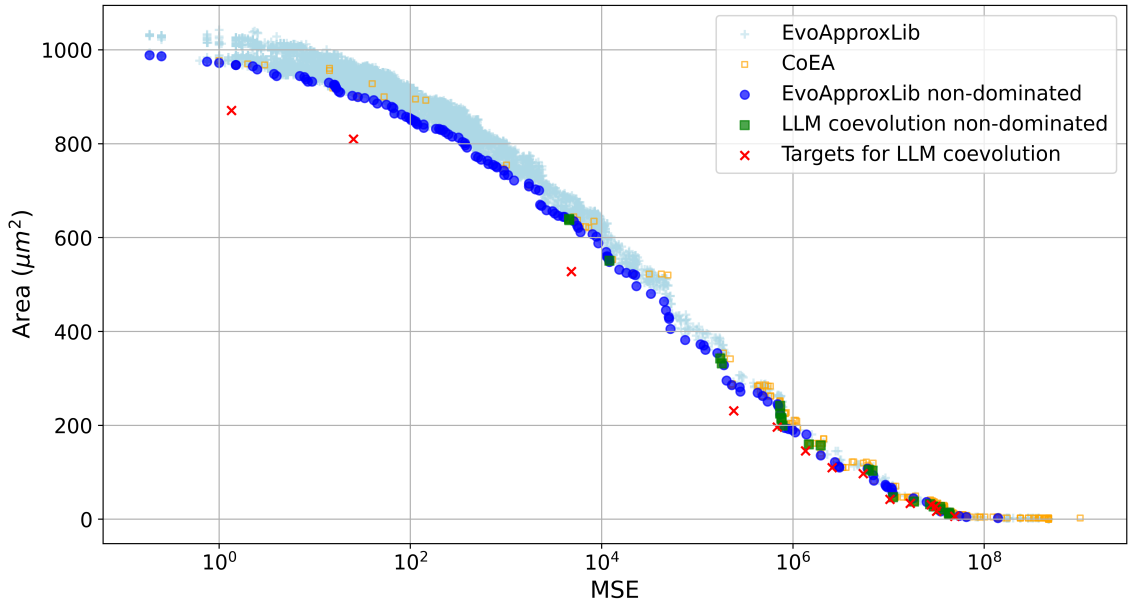


Figure 6.17: The *MSE-Area* trade-offs for 8-bit approximate multipliers generated by the proposed CoEA or taken from EvoApproxLib. The united global Pareto front contains non-dominated solutions generated by CoEA (green squares) and taken from EvoApproxLib (blue dots). The remaining points are dominated. The target specifications provided to CoEA are depicted with red crosses.

6.7.4 Synthesis

The results of the long-run experiments were converted into Verilog using a Python script. The Verilog files were tested by the iVerilog[1] tool for format correctness. Then the synthesis was performed. A colleague from the supervisor's team helped us with the synthesis and provided reference datasets. The results are shown in Figure 6.18 for WCE and Figure 6.19 for MSE trade-offs. We compare the CoEA-generated circuits to the Pareto fronts of the source dataset, Alberta2, public (cgp-evoapproxlib), and private (cgp-approx14, cgp-approx14ep) EvoApproxLib. The datasets are represented by their Pareto fronts; only the CoEA displays all the data. We labeled 10 circuits from the Pareto front to compare the trade-offs across different metrics.

The circuits with area below 200 μm^2 are close to or on the global Pareto front for the error-area pair. For WCE, circuits with error 4000 and higher are near the global Pareto front. With a larger area, the synthesized circuits diverge from Pareto and exhibit a worse trade-off between area and error. For MSE, the breaking point is 10^6 and higher.

We observed other important parameters of the synthesized circuits. The observed trade-offs are between error and power consumption, or power-delay product (PDP). Power consumption is important because it directly influences battery life when the circuit is used in an embedded system. PDP captures the energy consumed per operation. PDP combines the two metrics, helping balance speed and low power consumption for edge devices. The error-power and error-PDP trade-offs are displayed in the middle and bottom of Figures 6.18 and 6.19. When comparing power and PDP with WCE, circuits with error 3000 and higher show reasonable trade-offs comparable to the Pareto of the reference datasets. For MSE pairs, circuits with errors 10^5 or higher are close to the reference datasets' Pareto fronts.

We selected 10 circuits from the MSE-area and 10 from the WCE-area Pareto fronts separately. These circuits are labeled and displayed with corresponding labels on each figure. We observe the circuits preserve order. The only exceptions are designs with labels 2 and 3. Design 3 shows a worse trade-off of PDP-MSE compared to design 2; we observed the same for PDP-WCE.

The gap between synthesized and estimated values may be due to several factors. One of the most likely ones is fan-out. In a logic representation, a gate's output may connect to an arbitrary number of inputs. The synthesis, however, must resolve these connections, which might increase the size. To be clear, the synthesized area is typically smaller than the estimated one. However, after synthesis, some generated non-dominated circuits slightly increase the area compared to the reference dataset, and they become dominated by the reference circuits. The EvoApproxLib reference dataset was created using the CGP technique described in Section 3.3. The CGP respects the structure of a circuit. This explanation also aligns with the observation that the larger the circuit area, the greater the gap.

6.8 Experiments summary

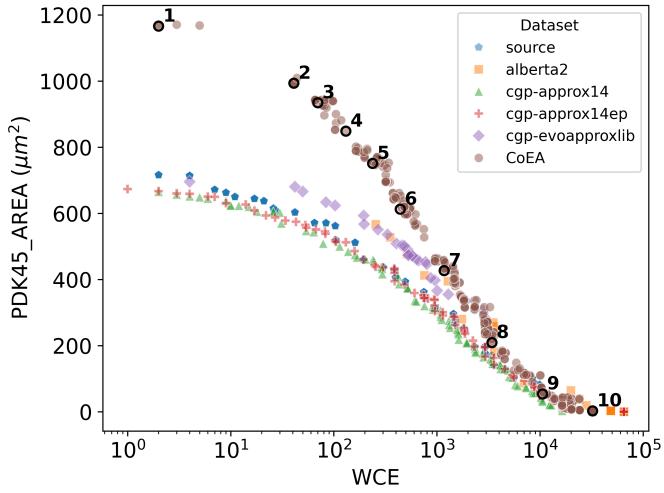
The experiments mostly explored the use of smaller 7B models for approximate multiplier design; the final experiments used large 120B and 480B models.

The initial experiments were focused on model selection. We tested four models and selected the most compatible model, Qwen2.5-Coder-7B-Instruct, with the equation format. This model was trained for coding tasks. The following experiments focused on the number of examples in the few-shot prompting with this model. We found that when an LLM is

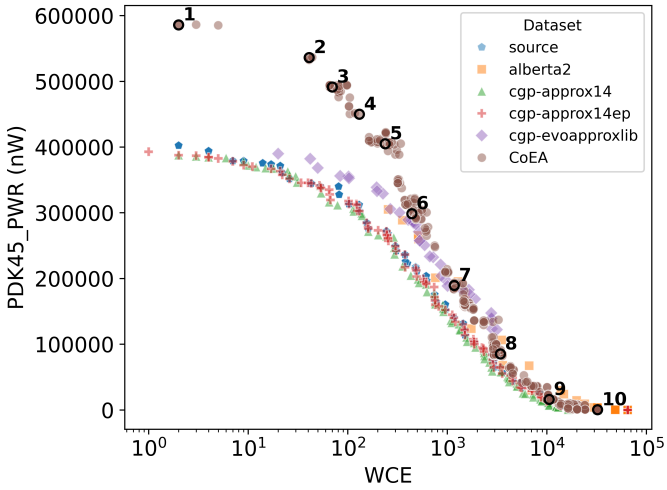
[1] https://github.com/steveicarus/iverilog

given 1 or 3 example circuits, it produces new circuit designs with the best parameters. The results of the experiments show that LLMs with few-shot prompting can generate valid, unseen designs without fine-tuning on the equation format. However, many syntactic and semantic errors occur with just a single call of a pretrained LLM, even with few-shot prompting. Fine-tuning is required to increase the likelihood of generating valid circuits.

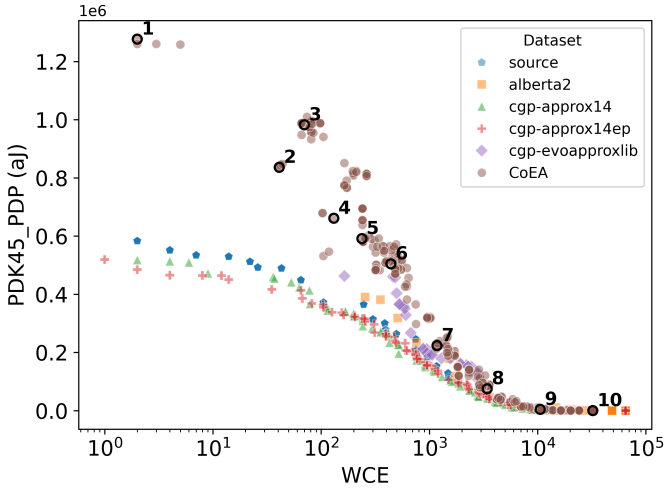
After the initial experiments, we fine-tuned the selected model. Not even a fine-tuned LLM is sufficient to surpass the EvoApproxLib circuits, even though one Pareto-dominant circuit was generated using a fine-tuned model with a setting temperature of 0.4. To further improve results, we tested fine-tuned models integrated in a loop and an evolutionary algorithm. The circuits obtained from these experiments show a tendency towards extreme compromises. They have either a small area and high error or vice versa. We finished the experiments with a combination of coevolution and significantly larger models. The resulting circuits successfully reached the Pareto front and slightly improved it in the error-area estimations.



(a) Area-WCE

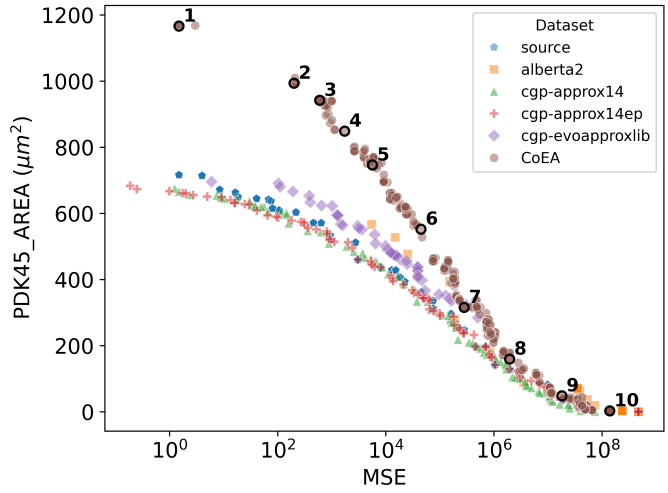


(b) Power-WCE

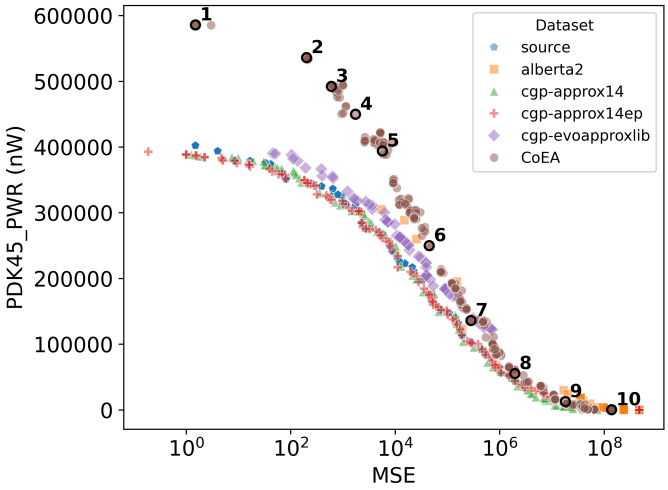


(c) Power Delay Product-WCE

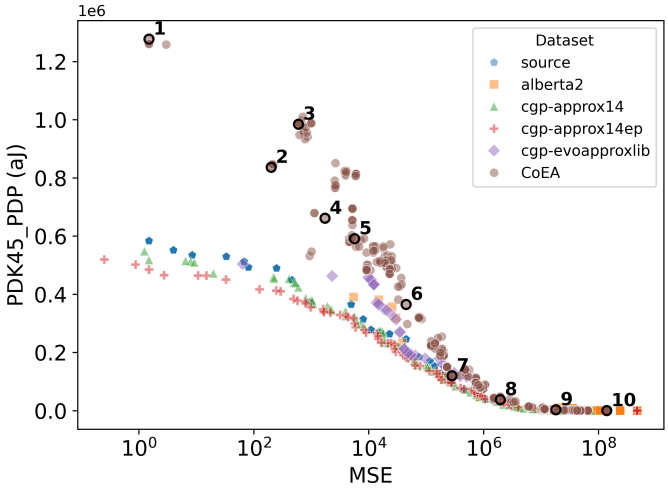
Figure 6.18: Synthesis of circuits produced by CoEA for WCE. The proposed designs are compared to reference datasets.



(a) Area-MSE



(b) Power-MSE



(c) Power Delay Product-MSE

Figure 6.19: Synthesis of circuits produced by CoEA for MSE. The proposed designs are compared to reference datasets.

Chapter 7

Conclusion

The main goal of this thesis was to propose a method for approximate circuit optimization utilizing LLMs. For that purpose, existing methods were studied, and initial experiments were conducted to further explore this topic before designing more complex solutions. The final solution was constructed iteratively based on preceding experiments.

We proposed an equation format for representing the approximate circuits. That format was tested across multiple LLMs to identify the most suitable model for future work. We then conducted experiments with few-shot prompting to determine how many example circuits LLM needs to generate target designs more precisely. We also proposed a layer inversion technique that transforms the output of a single layer of the neural network. The technique showed positive results for fine-tuned models, mainly without the refinement loop.

We fine-tuned a 7B LLM for 8 epochs, then inserted it into a refinement loop and an evolutionary algorithm. These experiments produced higher-quality circuits but also revealed limitations in achieving balanced trade-offs in the middle region of the error-area space.

The final experiments used models with significantly more parameters without further fine-tuning. These out-of-the-box models can utilize the knowledge from pretraining for circuit generation. Simply prompting these models was insufficient, which motivated the design of a coevolutionary algorithm that evolves prompt templates and circuits together. Experiments showed that the proposed CoEA method can find 8-bit approximate multiplier designs with estimated WCE-area and MSE-area trade-offs not present in EvoApproxLib, with 25 designs reaching the global Pareto front for the WCE objective and 23 for the MSE objective.

There are multiple directions for future work. The gap between estimated and synthesized area should be addressed, for example, by incorporating lightweight synthesis feedback into the fitness function or by adjusting the format. Fine-tuning LLMs used in coevolution on domain-specific data may further improve the quality of the generated circuits. Agentic systems have proven effective for software generation and could be adapted here by defining tools for circuit measurement and manipulation. To improve efficiency in longer runs, the distillation of large LLMs used in coevolution is a promising approach. For future work with smaller fine-tuned models, reinforcement learning may be a more effective training strategy than supervised fine-tuning. Finally, the layer inversion technique proposed in this work deserves further investigation, potentially across different tasks.

The results of this thesis were presented at Excel@FIT 2026. The results demonstrate that combining LLMs with coevolutionary algorithms is a viable approach for generating approximate circuit designs that discover high-quality, valid circuit implementations.

Bibliography

[1] Armeniakos, G.; Zervakis, G.; Soudris, D. and Henkel, J. Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey. *ACM Comput. Surv.*, 2023, vol. 55, no. 4, p. 83:1–83:36.

[2] Bahrami, M.; Mansoorizadeh, M. and Khotanlou, H. Few-shot Learning with Prompting Methods. In: *2023 6th International Conference on Pattern Recognition and Image Analysis (IPRIA)*. 2023, p. 1–5.

[3] Blank, J. and Deb, K. Pymoo: Multi-Objective Optimization in Python. *IEEE Access*, 2020, vol. 8, p. 89497–89509.

[4] Ceska, M.; Matyas, J.; Mrazek, V.; Sekanina, L.; Vasicek, Z. et al. Approximating Complex Arithmetic Circuits with Formal Error Guarantees: 32-bit Multipliers Accomplished. In: *Proc. of 36th IEEE/ACM Int. Conf. On Computer Aided Design*. IEEE, 2017, p. 416–423. ISBN 978-1-5386-3093-8.

[5] Chandrasekharan, A.; Soeken, M.; Grosse, D. and Drechsler, R. Precise error determination of approximated components in sequential circuits with model checking. In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, p. 1–6.

[6] Chippa, V.; Mohapatra, D.; Roy, K.; Chakradhar, S. and Raghunathan, A. Scalable Effort Hardware Design. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, september 2014, vol. 22, p. 2004–2016.

[7] Deb, K.; Pratap, A.; Agarwal, S. and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 2002, vol. 6, no. 2, p. 182–197.

[8] Dettmers, T.; Pagnoni, A.; Holtzman, A. and Zettlemoyer, L. QLORA: efficient finetuning of quantized LLMs. In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2023. NIPS '23.

[9] Devlin, J.; Chang, M.-W.; Lee, K. and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: Burstein, J.; Doran, C. and Solorio, T., ed. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, p. 4171–4186. Available at: https://aclanthology.org/N19-1423/.

[10] Drahosova, M.; Sekanina, L. and Wiglasz, M. Adaptive Fitness Predictors in Coevolutionary Cartesian Genetic Programming. *Evolutionary Computation*, 2019, vol. 27, no. 3, p. 497–523.

[11] Hrbacek, R.; Mrazek, V. and Vasicek, Z. Automatic design of approximate circuits by means of multi-objective evolutionary algorithms. In: *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2016, p. 1–6.

[12] Hu, E. J.; Shen, Y.; Wallis, P.; Allen Zhu, Z.; Li, Y. et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. Available at: https://arxiv.org/abs/2106.09685. ArXiv preprint.

[13] Jiang, H.; Liu, L.; Lombardi, F. and Han, J. Approximate Arithmetic Circuits: Design and Evaluation. In: Reda, S. and Shafique, M., ed. *Approximate Circuits, Methodologies and CAD*. Springer, 2019, p. 67–98.

[14] Jiang, H.; Santiago, F. J. H.; Mo, H.; Liu, L. and Han, J. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proceedings of the IEEE*, 2020, vol. 108, no. 12, p. 2108–2135.

[15] Kamath, U.; Graham, K. L. and Emara, W. *Transformers for Machine Learning: A Deep Dive*. Boca Raton: CRC Press, 2022. 19–27 p. Chapman & Hall/CRC Machine Learning & Pattern Recognition Series. ISBN 978-0-367-76734-1. Taylor & Francis Group.

[16] Kukkonen, S. and Deb, K. A Fast and Effective Method for Pruning of Non-dominated Solutions in Many-Objective Problems. In:. January 2006, p. 553–562. ISBN 978-3-540-38990-3.

[17] Kulkarni, P.; Gupta, P. and Ercegovac, M. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In:. February 2011, p. 346 – 351.

[18] Lehman, J.; Gordon, J.; Jain, S.; Ndousse, K.; Yeh, C. et al. Evolution Through Large Models. In: Banzhaf, W.; Machado, P. and Zhang, M., ed. *Handbook of Evolutionary Machine Learning*. Singapore: Springer Nature Singapore, 2024, p. 331–366. ISBN 978-981-99-3814-8. Available at: https://doi.org/10.1007/978-981-99-3814-8_11.

[19] Liu, Y.; Zhang, T. and Parhi, K. Computation Error Analysis in Digital Signal Processing Systems With Overscaled Supply Voltage. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, may 2010, vol. 18, p. 517 – 526.

[20] Madaan, A.; Tandon, N.; Gupta, P. et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. Accessed: 2025-11-01.

[21] Mrazek, V.; Hrbacek, R. et al. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In: *Proc. of DATE'17*. 2017, p. 258–261.

[22] Mrazek, V.; Sekanina, L. and Vasicek, Z. Libraries of Approximate Circuits: Automated Design and Application in CNN Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2020, vol. 10, no. 4, p. 406–418.

[23] Mrazek, V.; Vasicek, Z. and Sekanina, L. Evoapproxlib: Extended library of approximate arithmetic circuits. In: *Proceedings of the Workshop on Open-Source EDA Technology (WOSET)*. 2019, p. 10. Available at: https://woset-workshop.github.io/PDFs/2019/a10.pdf.

[24] Sekanina, L.; Vasicek, Z. and Mrazek, V. Inexact Arithmetic Operators. In: Bosio, A.; Ménard, D. and Sentieys, O., ed. *Approximate Computing Techniques: From Component- to Application-Level*. Cham: Springer International Publishing, 2022, p. 81–107. ISBN 978-3-030-94705-7. Available at: https://doi.org/10.1007/978-3-030-94705-7_4.

[25] Si, C.; Yang, D. and Hashimoto, T. *Can LLMs Generate Novel Research Ideas? A Large-Scale Human Study with 100+ NLP Researchers*. 2024. Available at: https://arxiv.org/abs/2409.04109. ArXiv preprint.

[26] Stanley Marbell, P.; Alaghi, A.; Carbin, M.; Darulova, E.; Dolecek, L. et al. Exploiting Errors for Efficiency: A Survey from Circuits to Applications. *ACM Comput. Surv.* New York, NY, USA: ACM, 2020, vol. 53, no. 3. ISSN 0360-0300.

[27] Su, S.; Meng, C.; Yang, F.; Shen, X.; Ni, L. et al. VECBEE: A Versatile Efficiency–Accuracy Configurable Batch Error Estimation Method for Greedy Approximate Logic Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022, vol. 41, no. 11, p. 5085–5099.

[28] Subramani, N.; Suresh, N. and Peters, M. Extracting Latent Steering Vectors from Pretrained Language Models. In: Muresan, S.; Nakov, P. and Villavicencio, A., ed. *Findings of the Association for Computational Linguistics: ACL 2022*. Dublin, Ireland: Association for Computational Linguistics, May 2022, p. 566–581. Available at: https://aclanthology.org/2022.findings-acl.48/.

[29] Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Dolan Gavitt, B. et al. VeriGen: A Large Language Model for Verilog Code Generation. *ACM Trans. Des. Autom. Electron. Syst.* New York, NY, USA: Association for Computing Machinery, april 2024, vol. 29, no. 3. ISSN 1084-4309. Available at: https://doi.org/10.1145/3643681.

[30] Tomašovič, M. *Large Language Models in Speech Recognition*. Brno, 2024. Bachelor's Thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Beneš, K. Available at: https://www.vut.cz/en/students/final-thesis/detail/153477. Available online; accessed 2026-01-04.

[31] Turner, R. E. *An Introduction to Transformers* arXiv (Cornell University) [online]. 2023. Available at: https://doi.org/10.48550/arXiv.2304.10557. Published: 2023-04-20; Accessed: 2026-01-03.

[32] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L. et al. Attention is All you Need. In: Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available at: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[33] Xiao, W.; Putrevu, V. S. C.; Hemadri, R. V.; Garg, S. and Karri, R. *PrefixLLM: LLM-aided Prefix Circuit Design* arXiv (Cornell University) [online]. 2024. Available at: https://doi.org/10.48550/arXiv.2412.02594. Accessed: 2026-01-01.

[34] Yi, S.; Zuo, W.; Wu, H.; Dai, R.; Qian, W. et al. GPTAC: Domain-Specific Generative Pre-Trained Model for Approximate Circuit Design Exploration. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2025, vol. 15, no. 2, p. 349–360.

[35] Zhang, Y.; Khan, S. A.; Mahmud, A. K. M. F. et al. Exploring the Role of Large Language Models in the Scientific Method: From Hypothesis to Discovery. *Npj Artificial Intelligence*, 2025, vol. 1, no. 1. ISSN 3005-1460. Online; Published: 2025-08-05; Accessed: 2026-01-11.

Appendix A

Experiment results

Many experiments were conducted within this thesis. This appendix presents informative results that were omitted from the main text to keep it clear.

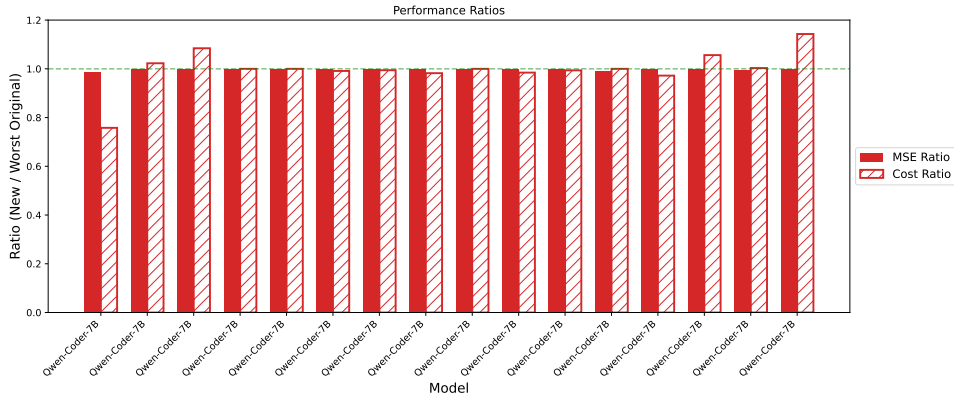
A.1 Few-shot model comparison

Figure A.1 shows complementary data to the ones in Figure 6.1. It shows the ratio between the maximal values of MSE and cost for the same dataset samples. The graphs further confirm the Qwen2.5-Coder-7B-Instruct dominance over the rest of the selected models.

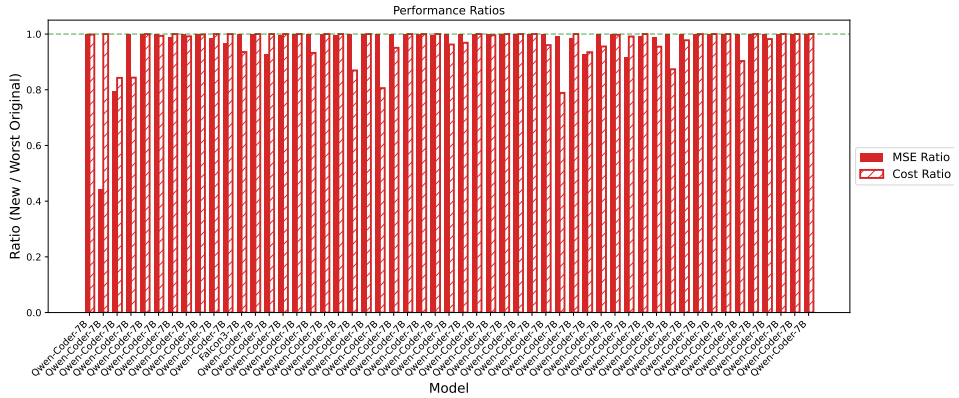
The overall circuits generated from prompt versions 1 and 2 are displayed in Figure A.2. The diversity increased significantly with version 2, and the circuits cover a larger range of MSE-area trade-offs.

A.2 Fine-tuned model experiments results

We present a graph displaying MSE-area Pareto fronts for temperatures from 0.0 to 2.0 (upper left Figure A.3). A Pareto front for inversion at each layer of the tested model is portrayed in the upper right Figure A.3. The bottom graphs of Figure A.3 show the overall distribution of generated and existing data, and the graph on the right even captures all generated circuits to display their distribution.

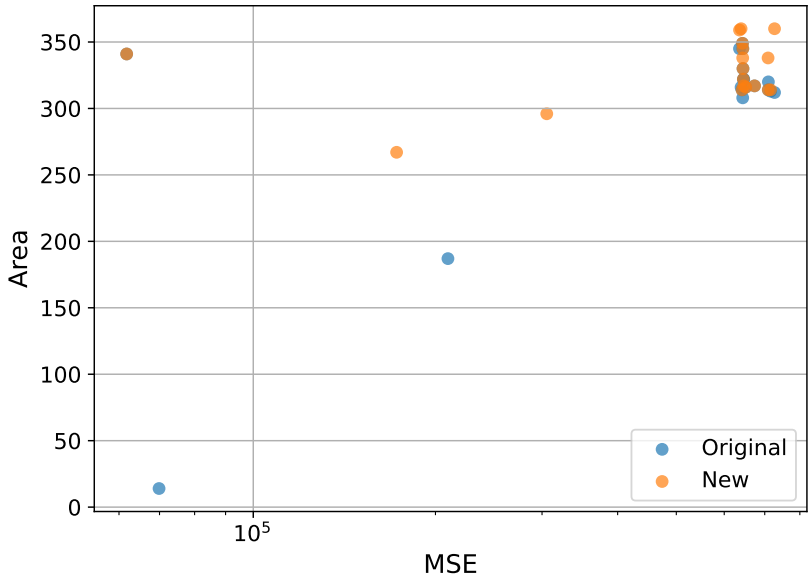


(a) Prompt 1 MSE and cost ratios

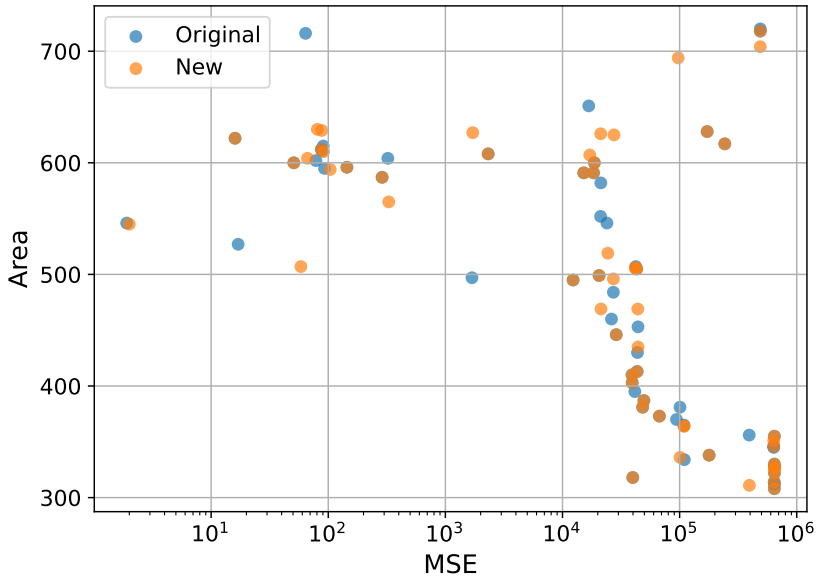


(b) Prompt 2 MSE and cost ratios

Figure A.1: Results of few-shot prompting. The prompt contains two example circuits; the target is to minimize MSE while preserving area (cost). The data in these graphs are filtered to fit the condition that the MSE of the generated circuit is lower or maximally 10% higher than the minimal MSE of the circuit from the prompt, and the same condition for cost. Columns express ratios between the maximal MSE from the given two example circuits and the MSE of the generated circuit, similarly for cost. Part a) shows the results of prompting with the prompt template version 1, and Part b) shows version 2.



(a) Prompt 1 MSE and cost ratios



(b) Prompt 2 MSE and cost ratios

Figure A.2: A scatter plot showing MSE to cost comparison. The data in these graphs are filtered to fit the condition that the MSE of the generated circuit is lower or maximally 10% higher than the minimal MSE of the circuit from the prompt, and the same condition for cost. Part a) shows the results of prompt template version 1, and Part b) shows version 2.

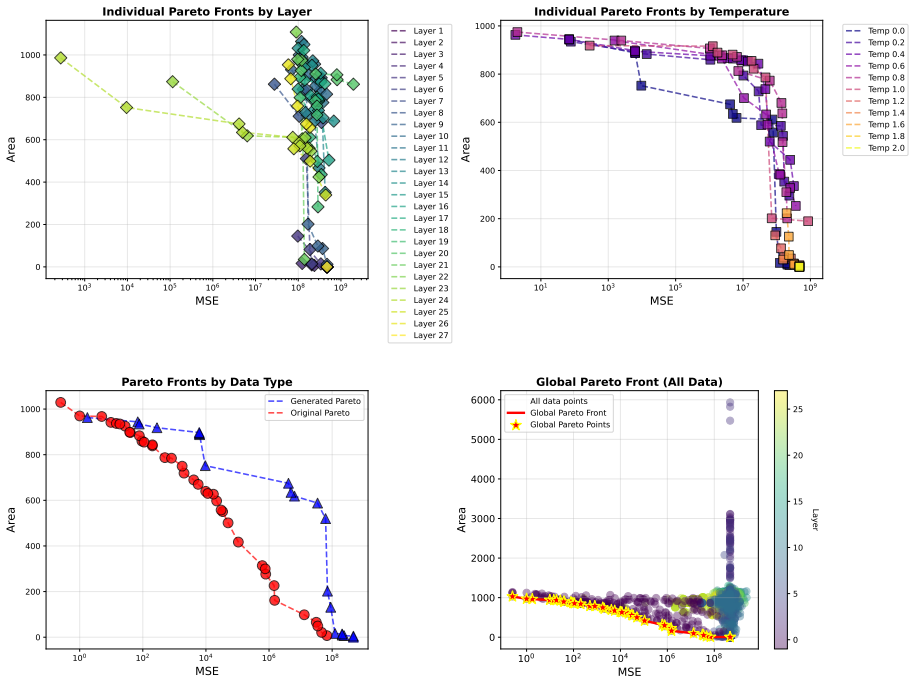


Figure A.3: Comparison of Pareto fronts for each setting individually.