LANGUAGE MODELS FOR PROGRAM REASONING AND OPTIMIZATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Anjiang Wei
March 2026

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken, Primary Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Clark Barrett

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Azalia Mirhoseini

Approved for the Stanford University Committee on Graduate Studies.

Kenneth Goodson, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Large language models (LLMs) have reshaped how developers write software. They now act as coding assistants capable of generating code directly from natural language specifications, a practice known as "vibe coding." This thesis pursues two core directions: (1) extending LLMs beyond code assistance to improve the performance of software systems, and (2) investigating the limitations of current LLMs in reasoning about programs.

I present three applications of LLMs in software systems aimed at performance optimization. First, I show how LLMs can improve the performance of parallel programs in the Legion programming system by generating programs in a high-level domain-specific language that controls how computation and data are mapped onto the machine. This agent-system interface provides a clean abstraction that enables LLMs to act as optimizers through code generation. Second, beyond parallel programs, I demonstrate that LLMs can serve as superoptimizers for assembly code, achieving performance that exceeds state-of-the-art compiler optimizations. This is done using reinforcement learning with a reward that jointly captures correctness and speedup. However, such LLM-generated assembly lacks correctness guarantees, underscoring the need for verification techniques. Third, to make verification more scalable, I use LLMs to synthesize invariants that accelerate solver performance by decomposing the original assertion into two simpler verification queries. I also study how supervised fine-tuning and Best-of-N sampling further improve the efficiency of the verification.

On a complementary line of work, I develop methods to expose and study the limitations of current LLMs in program reasoning. I focus on two classical challenges in programming languages: equivalence checking and program synthesis. Equivalence checking asks whether two programs produce identical outputs for all possible inputs; we use it to assess how well LLMs understand program behavior. For program synthesis, the model must produce a program that satisfies given input–output examples and generalizes to unseen inputs. We design an evaluation protocol that mirrors real-world reverse-engineering scenarios while testing the inductive reasoning capabilities of LLM-based agents.

Together, these two directions represent early explorations of how LLMs can enhance software systems while also identifying open challenges in advancing their ability to reason about programs.

Acknowledgments

Standing at the end of my PhD, I am filled with gratitude for the many people whose support carried me through this journey.

First and foremost, I would like to thank my advisor, Alex Aiken, for his tremendous support in both my research and my personal development. I am deeply grateful that Alex chose to take me on as his student. Despite my limited prior background in programming languages and high-performance computing, he was willing to take a chance on me and gave me the opportunity to align with him. I have long appreciated his confidence in me.

I am deeply grateful to Alex for helping me grow into an independent researcher. Although I was exposed to research as an undergraduate, much of that experience involved working within directions set by senior PhD students or faculty, which gave me a sense of safety as an executor. Alex's mentorship was fundamentally different. Early on, he told me, "You are supposed to be thinking about what project to do next," and guided me through choosing research directions. He emphasized, "I want you to own the project and make the important decisions." That transition was difficult. During this period, I discontinued several projects and experienced a series of paper rejections. Yet Alex never lost faith in me or in our work. Instead, he remained consistently supportive and encouraged me to think deeply about how to improve the research, even when reviewer feedback was discouraging. These experiences shaped me into a researcher who can make independent decisions, trust my own judgment, and remain optimistic and persistent in the face of setbacks.

I would also like to thank Alex for his broad research vision, which gave me both the freedom and the opportunity to explore a diverse range of topics during my PhD. When I expressed my interest in program verification and my desire to work with Caroline Trippel on a new direction, Alex was supportive. This allowed me to explore and appreciate the elegance of verification in microarchitecture design. Later, I was given the valuable opportunity to collaborate with Hang Song, through which I learned many interesting topics in computational fluid dynamics, including the foundational Navier-Stokes equations. Alex has also been supportive of my research interests in large language models. Throughout my PhD, Alex has consistently encouraged me to pursue my intellectual curiosities and supported me at every step. His openness, trust, and encouragement have played a central role in shaping my growth as a researcher.

I also want to thank Alex for showing me what true love for a field and perseverance in research truly mean, qualities that I deeply admire. Alex has worked on Legion for more than a decade, yet he remains genuinely excited about it. Even though task-based programming is a relatively small area within the high-performance computing community and has not been the most popular programming paradigm yet, Alex never stopped making progress on Legion. He continued to refine and advance Legion with full commitment, ultimately making it the most mature and capable task-based programming system to date, far ahead of contemporaneous efforts. His passion for long-term research has left a profound impression on me and taught me the true power of vision, patience, and perseverance.

Last but not least, I would like to thank Alex for his philosophy on life and for setting a great example of how to live with balance. I will always remember his advice to me: "It's good to be ambitious, but don't let your desire eat you." Alex works intensely during the academic term, yet he consistently makes time for his family and for travel during breaks. He also reminded me to slow down and take care of myself when I was under too much stress. Beyond research, I often brought philosophical questions about research and life into our meetings, and I am deeply grateful that Alex was always willing to share his perspective and wisdom.

I would also like to thank Clark Barrett and Azalia Mirhoseini for serving on my reading committee. I am grateful to Clark for taking me on as a rotation student and introducing me to research topics in verification. During my rotation with him, I enjoyed engaging with the beauty of theory and made friends with several members of his group, which planted the seed for the chapter on invariant synthesis in this thesis. I am thankful to Azalia for her inspiring work on applying AI to address challenges in systems. Reading her research papers helped me think about problems in software systems from a different perspective.

I would like to thank Caroline Trippel and Priyanka Raina for serving on my oral exam committee. I am grateful to Caroline for giving me the opportunity to work on a project involving reentry analysis for microarchitecture design, where I was exposed to topics in verification, security, and hardware design, all of which I had long been interested in despite my limited relevant background in undergraduate studies. I would also like to thank Priyanka Raina, Sara Achour, and Subhasish Mitra for guiding my first research project at Stanford on resistive random access memory (RRAM). These projects helped satisfy my long-standing curiosity about how hardware works, a domain I had wondered about for many years and was finally able to explore and contribute to during my PhD.

I would like to sincerely thank Ke Wang for his collaboration. We spent a great deal of time discussing research ideas, and he generously helped with paper writing and stayed up late with me to meet deadlines. I am also grateful for the many insightful research discussions with other collaborators, including my internship mentors Saeed, Abhinav, Zhenyu, and Shaowei. I also had the opportunity to serve as a teaching assistant for CS224N, taught by Diyi and Tatsunori. I would like to thank Diyi for the research discussions we shared. Through the course, I was given the valuable

opportunity to mentor students on their research projects, during which I learned how to evaluate project ideas and provide constructive feedback. Also, I am very thankful to the intern students I had the privilege to work with, especially Tarun, Tianran, Yuheng, Huanmi, Jiannan, Naveen, Ran, and Yogesh. Discussing research problems with junior students has been fun and has reminded me of the strong research passion and curiosity that motivated me to apply to PhD programs.

I thank Matthew, Shiv, Scott, Chris, Olivia, Ziheng, Allen, Thiago, Zach, Rohan, David, AJ, Rupanshu, Gina, Yao, Sam, Anna, Genghan, Qizheng, Daniel, Yicheng, Konstantin, Nathan, Rubens, Jungwoo, Yuneng, Rylan, Rachel, Luke, Yuhui, Xiaohan, Mert, Ying, Mofan, Zhiqiao, Miles, Joydeb, John, and Shirley for the many conversations and shared experiences that made my PhD journey so much happier. I will always cherish the English slang that Chris patiently tried to teach me, the ping-pong and foosball games I played with Matthew, Thiago, and Hang, and the long hours Matthew and I spent discussing research problems and research philosophy. We both loved the idea that conducting research is like "planting the little acorns from which the mighty oak trees grow." I will always remember the countless weekends I spent working with Hang, whose dedication and deep immersion in research set a remarkably high standard for me. Also, I will never forget Olivia's comforting words, "life is just a collection of experiences," after my paper was rejected five times in a row. I will always remember the ice skating experience with Shiv, meditation sessions with Scott, the birthday celebrations of David and Rohan, the Yosemite trip with Yuneng, the Lake Tahoe trip with Tony, the Arizona trip with Yuhui, the piano practice sessions with Mofan and Zhiqiao, the unforgettable defense celebration with costumes from Sam, Anna, Gina, and many other friends, and the countless small moments that made these years deeply meaningful to me.

I would also like to thank my advisors during my undergraduate studies: Darko Marinov, Lingming Zhang, Tao Xie, and Yun (Eric) Liang. I am deeply grateful to Darko for his mentorship. His passion for research left a lasting impression on me and strongly motivated me to pursue a career in research. I would like to thank Lingming for his broad research vision, his guidance on developing my writing skills, and his support during my early steps toward research independence. I am thankful to Tao for his thoughtful advice on choosing graduate schools and advisors, which guided me at a crucial moment in my academic path. I am especially grateful to Yun for introducing me to research when I joined his lab by encouraging me to read broadly, including Alex's TASO paper. Yun urged me to pursue research in the spirit of Alex, and I am deeply thankful for his guidance and encouragement, which played an important role in shaping my path.

Finally, I would like to thank my family for their unconditional support. I am deeply thankful to my parents for their constant love and support throughout the ups and downs of my PhD journey. I am also grateful to my grandparents for their care. Even from far away, their love has always been with me, and it has meant more to me than I can express.

Contents

Abstract iv

Acknowledgments v

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Language Models for Performance Optimization . . . 1
- 1.3 Limitations in Language Models . . . 4
- 1.4 Roadmap . . . 5

2 Optimizing Parallel Programs with Language Models 6
- 2.1 Introduction . . . 6
- 2.2 Related Work . . . 8
- 2.3 Problem Definition . . . 9
- 2.4 Our Approach: Agent-System Interface . . . 10
  - 2.4.1 Domain-Specific Language Design . . . 10
  - 2.4.2 Generative Optimization via AutoGuide . . . 13
- 2.5 Evaluation . . . 15
  - 2.5.1 Speedup of Application Performance . . . 15
  - 2.5.2 Ablation Study of DSL for Code Generation . . . 17
  - 2.5.3 Ablation Study of the AutoGuide Feedback . . . 18
- 2.6 Conclusion . . . 19

3 Optimizing Assembly Programs with Language Models 20
- 3.1 Introduction . . . 20
- 3.2 Related Work . . . 22
- 3.3 Methodology . . . 23
  - 3.3.1 Task Definition . . . 23
  - 3.3.2 Dataset Construction . . . 24

3.3.3 Reinforcement Learning . . . 25
3.3.4 Best-of-N Sampling, Supervised Fine-Tuning, and Iterative Refinement . . . 26
3.4 Experimental Setup . . . 27
3.5 Results . . . 28
3.5.1 Evaluation of Different Models . . . 28
3.5.2 Effectiveness of RL Training . . . 30
3.5.3 Results from Supervised Fine-Tuning and Inference-Time Methods . . . 30
3.5.4 Analysis of Program Transformations . . . 31
3.6 Discussion . . . 32
3.7 Conclusion . . . 35

4 Accelerating Verification via Invariant Synthesis 36
4.1 Introduction . . . 36
4.2 Related Work . . . 38
4.3 Method . . . 39
4.3.1 Preliminary . . . 39
4.3.2 Verifier-Based Algorithm . . . 40
4.3.3 Implementation . . . 41
4.3.4 Supervised Fine-Tuning and Best-of-N Sampling . . . 42
4.4 Experimental Setup . . . 43
4.5 Results . . . 45
4.5.1 Results of LLMs . . . 45
4.5.2 Comparison with Prior LLM-based Verifiers . . . 46
4.5.3 Failure Mode Analysis . . . 48
4.5.4 Results of Fine-Tuning and Best-of-N Sampling . . . 49
4.6 Conclusion . . . 50

5 Evaluating Program Reasoning via Equivalence Checking 51
5.1 Introduction . . . 51
5.2 Related Work . . . 53
5.3 Benchmark Construction . . . 54
5.3.1 Pairs from Program Analysis (DCE) . . . 54
5.3.2 Pairs from Compiler Scheduling (CUDA) . . . 56
5.3.3 Pairs from a Superoptimizer (x86-64) . . . 56
5.3.4 Pairs from Programming Contests . . . 58
5.4 Experimental Setup . . . 59
5.5 Results . . . 59
5.5.1 Model Accuracy . . . 59

5.5.2 Difficulty Analysis . . . 60
5.5.3 Bias in Model Prediction . . . 62
5.5.4 Prompting Strategies Analysis . . . 62
5.6 Discussion and Future Directions . . . 63
5.7 Conclusion . . . 65

6 Evaluating Inductive Reasoning via Program Synthesis 66
6.1 Introduction . . . 66
6.2 Related Work . . . 68
6.3 Method . . . 69
6.3.1 Problem Definition of Inductive Program Synthesis . . . 69
6.3.2 Interactive Evaluation Protocol for LLM Agents . . . 70
6.3.3 Benchmark Preparation . . . 71
6.3.4 Fine-Tuning on Synthetic Data . . . 71
6.4 Experiment Setup . . . 72
6.5 Results . . . 73
6.5.1 Main Results . . . 73
6.5.2 Do Initial Input-Output Examples Underspecify the Target Function? . . . 74
6.5.3 Ablation Study: Input-Output Queries and Oracle Feedback . . . 75
6.5.4 Performance of Fine-Tuned Models . . . 76
6.5.5 Case Study . . . 77
6.6 Discussion . . . 77
6.6.1 Results on BigCodeBench for Broader Domain Coverage . . . 77
6.6.2 Impact of Providing Access to a Python Interpreter . . . 78
6.7 Conclusion . . . 78

7 Conclusions 79
7.1 Summary . . . 79
7.2 Future Research Directions . . . 79

Bibliography 82

List of Tables

2.1 AutoGuide Feedback Mechanism. The AutoGuide mechanism interprets raw execution output from the runtime system, providing more informative error explanations and suggestions for mapper modifications. It is implemented via keyword matching. . . . 12

2.2 Code Generation Success Rates. Success rates for generating code across 10 mapping strategies described in natural language. The test evaluates whether the generated code compiles and passes execution tests. Generating DSL code significantly outperforms generating C++ for both settings. Symbols indicate results: – fails to compile, ✗ compiles but fails the test, and ✓ passes the test. 16

3.1 Dataset statistics across training and evaluation splits. LOC = lines of code. 27

3.2 Comparison of LLMs on our assembly optimization benchmark. We report compilation success rate, test pass rate, and average speedup over the gcc -O3 baseline. Speedup is averaged across all test inputs, with each input evaluated over ten runs. 29

3.3 Performance of the base model and the models trained with RL or supervised fine-tuning. Results include compilation success, test pass rates, and average speedup, reported over 5 runs with 95% confidence intervals. 30

3.4 Comparison of 0-shot, 2-shot, and 4-shot prompting across different models. 33

4.1 Dataset statistics. 43

4.2 Performance of different LLMs when using Quokka. We report the number of instances with verified-correct invariants and the number of instances achieving speedups of at least 1.2×, 1.4×, 1.6×, 1.8×, and 2.0× over UAutomizer. 44

4.3 Comparison of speedup performance across different methods. We report the number of instances achieving speedups of at least 1.2×, 1.4×, 1.6×, 1.8×, and 2.0× over UAutomizer. 46

4.4 Failure mode breakdown for the top three models on Quokka. *Incorrect Invariants*: the invariant is refuted by the solver. *Goal Timeout*: the invariant is verified, but proving the assertion under it times out. *Invariant Timeout*: verifying the invariant itself times out. *Both Timeout*: both verification queries time out. 48

4.5 Performance comparison of base and fine-tuned Qwen2.5-7B. Fine-tuning leads to modest improvements in both correctness and speedup metrics. 48

5.1 Statistics of the EquiBench dataset. 59
5.2 Accuracy of 19 models on EquiBench under 0-shot prompting. We report accuracy for each of the six equivalence categories along with the overall accuracy. 60
5.3 Accuracies on equivalent and inequivalent pairs in the CUDA and x86-64 categories under 0-shot prompting, showing that models perform significantly better on inequivalent pairs. Random guessing serves as an unbiased baseline for comparison. 63
5.4 Accuracies of different prompting techniques. We evaluate 0-shot and 4-shot in-context learning, both without and with Chain-of-Thought (CoT). Prompting strategies barely improve performance. 63

6.1 Number of functions and lines of code statistics for each benchmark source across both dataset versions. 72
6.2 Success rates of LLMs on CodeARC using both annotated and anonymized datasets. We also report the average number of observed input-output examples and oracle invocations. All open-source models are instruction-tuned. 73
6.3 Number of problems (in both datasets) where the synthesized function passes the initial examples compared to the oracle. 75
6.4 Success rates (%) with varying budgets on the observable input-output examples (on both datasets). 75
6.5 Success rates of LLaMA-3.1-8B-Instruct and its fine-tuned variant on annotated and anonymized datasets. Fine-tuning improves performance, especially on the annotated dataset. 76
6.6 Success rates (%) on the original CodeARC benchmark and BigCodeBench, which covers diverse domains such as scientific computing and machine learning, extending beyond algorithm-focused problems. 78
6.7 Success rates (%) with and without access to a Python interpreter. 78

List of Figures

2.1 Iterative mapper refinement with agent-based generative optimization. The system leverages the Agent-System Interface, which consists of the Domain-Specific Language (DSL) and AutoGuide. The DSL abstracts away the low-level system code, defining a search space for mapping strategies, while AutoGuide interprets execution results into actionable guidance. As iterations progress, the mapper evolves to improve performance. 7

2.2 Comparison of a DSL mapper and a C++ mapper. The DSL's declarative, high-level design abstracts away the complexity of low-level C++ code, serving as the core of the Agent-System Interface. The highlighted boxes illustrate how the same functionality, which requires extensive C++ system code, can be expressed concisely in just a few lines in DSL. 10

2.3 Agent Optimization Process. The mapper agent takes server specifications and application-specific information as input, generates mapper code, and executes it alongside the application on the server. Raw execution feedback is enriched using the AutoGuide mechanism, and the mapper is iteratively refined by an LLM optimizer to improve performance. 11

2.4 Performance Comparison. Normalized throughput for 9 benchmarks, comparing expert mappers, random mappers, the average optimization trajectories of Trace, OPRO, and OpenTuner in 10 iterations across 5 runs, and the best mappers found by Trace. 14

2.5 Comparison of Trace (generative optimizer) and OpenTuner (traditional RL) over 1K iterations (averaged across all 9 benchmarks). 14

2.6 Comparison of different feedback designs. 0-Shot and 5-Shot are baselines. Execution provides only the raw execution output as feedback. Explain provides additional explanations of execution errors. Suggest offers mapper modification suggestions. All feedback is automatically generated. 18

3.1 Overview of the assembly code optimization task. Given a C program and its baseline assembly from gcc -O3, an LLM is fine-tuned with PPO or GRPO to generate improved assembly. The reward function reflects correctness and performance based on test execution. 21
3.2 Best-of-N sampling results. 31
3.3 Iterative refinement results. 31
3.4 Categorization of the program transformations: Loop Restructuring (LR), Arithmetic Optimization (AO), Address Calculation (AC), Stack Canary Removal (SCR), Algorithmic Simplification (AS), Register Allocation (RA), Function Call Optimization (FC), Instruction Selection (IS), Branch Elimination (BE). 32
3.5 Case study comparing the C code, baseline assembly produced by gcc -O3, and optimized assembly generated by Claude-Opus-4. The model successfully replaces the loop with the specialized hardware instruction `popcnt`, resulting in a significantly more concise implementation. 33
3.6 Case study comparing the baseline assembly code snippet produced by gcc -O3 and the optimized assembly code snippet generated by Claude-Opus-4. Claude-Opus-4 eliminates the entry-path unconditional jump and alignment padding by fusing GCC's two-block loop into a single, simpler control-flow structure, calls `printf@PLT` directly (a simpler function variant without security checks), and removes gcc's stack-protector canary check. 34

4.1 Illustration of Quokka's evaluation pipeline. The LLM proposes an invariant by specifying a program location and predicate (e.g., location B with `x % 7 == 3`). The verification procedure then incorporates this invariant to prove the property `x != 700` using two verifier queries, and we measure the resulting speedup relative to a baseline without LLM assistance. 37
4.2 Number of instances solved by different methods over varying timeout budgets. . . . 48
4.3 Effect of Best-of-N sampling on the number of instances under different Fast_p settings. 49

5.1 Overview of EquiBench. We construct (in)equivalent program pairs from diverse sources, including C and CUDA programs, x86-64 assembly, and competitive programming, using automated transformations based on program analysis, compiler scheduling, superoptimization, and changes in algorithms or variable names. 52
5.2 An equivalent pair from the DCE category in EquiBench. In the left program, `c = 1` is dead code that has no effect on the program state and is removed in the right program. Such pairs are generated using the Dead Code Elimination (DCE) pass in compilers. 55

5.3 An equivalent pair from the CUDA category in EquiBench. Both programs perform matrix-vector multiplication ($y = Ax$). The right-hand program uses *shared memory tiling* to improve performance. Tensor compilers are utilized to explore different *scheduling strategies*, automating the generation. 55
5.4 An equivalent pair from the x86-64 category in EquiBench. Both programs are compiled from the same C function shown above, the left using a compiler and the right using a *superoptimizer*. The function counts the number of set bits in the input `%rdi` register and stores the result in `%rax`. Their equivalence has been formally verified by the superoptimizer. 57
5.5 Equivalent pairs from the OJ_A, OJ_V, OJ_VA categories in EquiBench. OJ_A pairs demonstrate *algorithmic equivalence*, OJ_V pairs involve *variable renaming* transformations, and OJ_VA pairs combine *both* types of variations. 58
5.6 Scaling Trend on EquiBench. Models exhibit consistent gains as parameters increase. 62

6.1 Overview of CodeARC. Our framework evaluates LLMs' reasoning capabilities in inductive program synthesis. The agent begins with input-output examples, interacts with a hidden target function via function calls, and uses a differential testing oracle to check the correctness of the synthesized function for self-reflection and refinement. 67
6.2 Scaling trend on CodeARC. 74
6.3 Success rates (%) of LLM models across varying numbers of oracle invocations. . . . 75
6.4 Case Study. The model queries edge cases, synthesizes a comparison function, receives a counterexample from the oracle, and corrects it with a set-based solution. 77

Chapter 1

Introduction

1.1 Background and Motivation

Large language models (LLMs) have fundamentally reshaped how developers write software, acting as coding assistants capable of generating code directly from natural language specifications, a practice known as "vibe coding" [34, 13, 123, 323, 128]. Today, many widely used tools, including Claude Code, Codex, Gemini CLI, and Cursor, leverage LLMs as interactive coding agents, enabling developers to write, modify, and refactor code through prompting. These tools have already demonstrated substantial gains in developer productivity.

Despite this progress, the role of large language models in addressing complex, real-world challenges in software systems remains controversial [2]. Some researchers question whether LLMs can meaningfully reason about the intricate behaviors of large systems, while others raise concerns about correctness and the absence of formal guarantees for LLM outputs [244]. These tensions motivate a set of pressing research questions: Can AI models move beyond surface-level code generation to support the design and implementation of complex software systems? What limitations do current LLMs exhibit when reasoning about programs, and how can these limitations be systematically exposed and studied? This dissertation pursues an ambitious agenda along two complementary directions. First, it extends LLMs beyond code assistance to improve the performance of software systems. Second, it rigorously characterizes the limitations of LLMs in program reasoning.

1.2 Language Models for Performance Optimization

Performance optimization has long been a central research topic in software systems. At its core, performance optimization consists of two key components. The first is the definition of an appropriate search space that captures the relevant optimization choices. The second is the design of effective heuristics to identify correct and high-performing candidates within that space. Different optimization

problems induce different search spaces, and the heuristics used to explore them also vary.

In practice, developers often rely on rule-based heuristics for performance optimization, such as those implemented in compilers [275] or embedded as default policies in high-performance runtime systems [21]. Although these heuristics are effective in many cases, they offer no guarantees of optimality and frequently leave performance potential untapped [30, 250, 221].

To more effectively navigate large and complex search spaces, many machine learning techniques have been proposed and adopted for performance optimization in systems. These include stochastic search methods such as MCMC sampling [216], tree-based models such as gradient-boosted decision trees [79], deep learning approaches [314, 319, 317, 220], and reinforcement learning methods [186, 10, 104, 105]. Together, these techniques have been applied to a broad class of optimization problems in systems, spanning chip floorplanning, compiler optimization, and SAT solving.

The emergence of large language models represents a pivotal shift in search techniques for performance optimization. Unlike prior machine learning approaches, which typically require task-specific training before they can perform meaningful predictions, LLMs are pretrained on vast amounts of data and encode a broad range of prior knowledge. As a result, they exhibit strong reasoning and coding capabilities without fine-tuning, including the ability to solve complex programming tasks. Moreover, LLMs can adapt to new tasks through in-context learning without additional training [25], often requiring only a few examples. Rather than operating as learned cost models that guide search indirectly [314, 317], LLMs can directly generate code to explore the space of candidate programs. Instead of relying on complex feature engineering, they can be prompted using natural language descriptions of optimization objectives together with the necessary contextual information. The resulting search process produces concrete candidate implementations, making optimization outcomes explicit and human interpretable through generated code. Because LLMs can be highly interactive, the search process can be made iterative and continuously refined using feedback from the execution environment. This ability to perform generative optimization through direct code generation while learning from environmental feedback distinguishes LLMs as optimizers from prior machine learning techniques and opens new opportunities for the design of LLM-integrated software systems.

Motivated by these capabilities, this dissertation examines three distinct applications of large language models in performance optimization. The first application focuses on improving the performance of parallel programs written in the Legion programming system. Specifically, we use LLMs to generate a separate program, called a mapper, which defines the policy for mapping distributed applications to hardware resources and plays a critical role in the application performance. The mapping interface and underlying runtime system are designed and implemented so that invalid mappers are rejected by the Legion runtime. As a result, even when an LLM generates incorrect code, the application does not silently produce incorrect results. Instead, execution fails with explicit error signals. To support effective code generation and optimization, we introduce an Agent-System Interface that simplifies mapper development and provides meaningful system feedback. At its core

is a domain-specific language that encapsulates all performance-critical mapping decisions while abstracting away low-level system complexity. The language defines a structured search space that enables systematic exploration of mapping strategies. To further support optimization, we design an AutoGuide mechanism that interprets raw execution outputs into informative and actionable guidance for LLM agents. Rather than relying solely on scalar performance rewards, our approach leverages richer feedback, including error explanations and concrete optimization suggestions expressed in natural language. Our experimental results show that mappers produced by LLM-based optimizers often surpass expert-written mappers, achieving substantial performance improvements across a range of benchmarks while reducing tuning time from days to minutes.

Beyond parallel programs, this dissertation demonstrates that large language models can also serve as effective superoptimizers for assembly code, achieving performance that exceeds state-of-the-art compiler optimizations. Existing superoptimization techniques typically focus on very short, straight-line assembly programs without loops and rely on CPU-based search heuristics that do not scale. To address these limitations, we construct the first large-scale dataset of assembly programs. Programs in this dataset average 130 lines and include loops, far exceeding the size and complexity of prior loop-free datasets, which typically contain fewer than 15 lines. Building on this dataset, we go beyond evaluating existing models and apply reinforcement learning to fine-tune large language models for superoptimization. We use a reward function that jointly captures functional correctness and performance speedup. The results show that our training substantially improves performance over pretrained models, with additional gains achieved through Best-of-N sampling and iterative refinement. This chapter constitutes the first application of reward-based reinforcement learning to LLMs for code performance optimization, in which correctness and execution speed are jointly encoded in the optimization objective.

The third application explored in this dissertation is the acceleration of program verification. As LLMs are increasingly used to generate software, providing formal guarantees that programs behave as intended becomes even more important. A long-standing challenge in program verification is the automatic discovery of loop invariants, namely conditions that must hold before and after each loop iteration. To effectively accelerate verification, such invariants must not only be correct but also sufficiently strong to prove the assertions. However, discovering strong invariants is difficult in general. In this chapter, we develop a principled and sound methodology for evaluating whether LLMs can generate invariants that meaningfully accelerate the verification process, especially given that LLM-generated invariants may be incorrect. Rather than checking whether LLMs produce the same invariants identified by existing tools, as in prior work, we introduce a verifier-based algorithm to determine the correctness of LLM-generated invariants and prove that this procedure is sound. Our results show that both supervised fine-tuning and Best-of-N sampling significantly improve model effectiveness in accelerating verification, establishing a new performance baseline. Using our algorithm, LLM-based approaches outperform state-of-the-art non-LLM-based solvers, whereas

prior LLM-based verification systems with substantially more complex designs rarely achieve such improvements.

1.3 Limitations in Language Models

Benchmarks play a central role in how progress in artificial intelligence is measured and interpreted. They enable principled comparisons between models and offer a systematic means of tracking advances in the field over time. By exposing systematic failure modes and unresolved weaknesses, benchmarks help identify limitations in existing approaches and guide future research directions. This perspective is essential for building reliable AI systems and for setting realistic expectations for their deployment in real-world applications [214]. For example, widely adopted benchmarks such as ImageNet [57], SQuAD [209], and SNLI [24] are large-scale datasets that have been credited as foundational drivers of progress in their respective research areas.

We next discuss the principles for designing effective benchmarks for LLMs [107]. First, a benchmark should be sufficiently challenging to meaningfully differentiate between models and to expose the limitations of current large language models. While this requirement may appear straightforward, it is difficult to satisfy in practice. For instance, HumanEval [34] emerged as a standard benchmark for code generation in 2021. However, accuracy on HumanEval for leading models has increased dramatically, rising from below 30% in 2021 to over 90% by 2024, rendering it largely saturated and no longer informative for measuring progress. This illustrates the need for benchmarks that remain challenging over time. Second, a good benchmark should be grounded in real-world use cases. A benchmark may be complex or diverse, but if it lacks relevance to practical applications, its ability to inform real progress is limited. The most valuable benchmarks evaluate capabilities that either directly matter to end users or serve as foundational building blocks for more complex real-world tasks. In software engineering, the examples are program repair [287, 286], testing [285, 59], bug resolution [295, 128], and program transpilation [303, 23], among others. Third, a benchmark should be designed to minimize data contamination [75]. Benchmark instances should be generated in a way that prevents models from having encountered them during training. Avoiding such contamination is essential to ensure that benchmark results reflect genuine generalization rather than memorization.

This dissertation focuses on exposing and systematically studying the limitations of LLMs by defining two new problem settings for evaluation. One benchmark introduced in this dissertation evaluates the ability of large language models to reason about program semantics. This task is fundamentally different from the code generation from natural language that has been the primary focus of most prior work. Instead, we investigate whether LLMs have a deeper understanding of program behavior. To study this question, we introduce program equivalence checking as a new evaluation task for reasoning about program semantics. Given two programs, equivalence checking

asks whether they are semantically equivalent, meaning that they produce identical outputs for all possible inputs, regardless of how different they are written. Program equivalence directly tests a model's ability to reason about program behavior, rather than surface-level code patterns. In fact, any question about program semantics can be formulated as an equivalence checking problem. Equivalence checking is undecidable in general, highlighting the intrinsic difficulty of this task. Moreover, equivalence checking has a broad practical use case. In real-world software development, developers frequently refactor code, optimize implementations, or translate programs across languages, and must determine whether functionality has been preserved. A language model more capable of reasoning about program equivalence would therefore support a wide range of software engineering tasks. Finally, our dataset is constructed through a fully automated pipeline, which allows the dataset to scale more easily while also reducing the likelihood of data contamination.

The other benchmark introduced in this dissertation is inspired by real-world scenarios in reverse engineering. In such settings, a reverse engineer observes the input-output behavior of a binary executable and seeks to synthesize a program that reproduces this behavior based solely on those observations. This problem is also known as inductive program synthesis in the programming languages community and directly probes the inductive reasoning capabilities of LLMs. Unlike prior work that evaluates models based on whether generated functions pass a fixed set of hidden test cases, our evaluation protocol more faithfully reflects real-world conditions. An LLM-powered agent begins with a small set of input-output examples, queries a hidden target function at adaptively chosen inputs, and receives feedback from a differential testing oracle to assess the correctness of its synthesized programs. This feedback loop enables self-reflection and iterative refinement, providing a principled and interactive framework for evaluating inductive reasoning in a more real-world setting for program synthesis.

1.4 Roadmap

The remainder of this dissertation is organized as follows. The chapters are based on several published papers and papers currently under submission. Chapter 2 presents the use of large language models for optimizing parallel programs [265]. Chapter 3 describes how LLMs can be applied to the optimization of assembly programs [269]. Chapter 4 explores the use of LLMs for invariant synthesis to accelerate program verification [268]. Chapter 5 introduces program equivalence checking as a benchmark for evaluating LLMs' capabilities in reasoning about program semantics [264]. Chapter 6 studies inductive reasoning in large language models through program synthesis [267]. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Optimizing Parallel Programs with Language Models

2.1 Introduction

Modern scientific discovery depends on advanced software tools for modeling and simulation [241, 261, 173]. Computational scientists, including physicists, chemists, and biologists, rely on high-performance computing to tackle complex problems. These scientific computations dominate workloads on the world's most powerful supercomputers [72]. However, many domain scientists lack expertise in computer science, and therefore having difficulties in optimizing their programs because of the complexity and scale of the underlying machines. Even for experts, finding and fixing performance problems resulting from program modifications or when porting to a new machine is often time-consuming. Any progress on automating performance tuning is of great benefit in this domain.

Task-based programming [238, 21, 12, 32, 190, 18] has emerged as a promising approach to high performance computing. The paradigm involves decomposing computations into independent *tasks* that communicate exclusively through their arguments. A key advantage of task-based systems is that the performance tuning problem is factored out into a separate *mapping*: an assignment of tasks to processors and data to particular memories. High-quality mapping, achieved through a well-designed *mapper* (implemented as code), can significantly improve performance, often by an order of magnitude [89].

However, currently writing mappers remains a labor-intensive process, as it requires deep knowledge of applications, hardware, and low-level system APIs. In addition, this process is highly application-specific, input-specific, and machine-specific, often taking experts several days of meticulous tuning to achieve high performance. This challenge is especially pronounced for domain scientists, who typically lack the necessary expertise in computer systems and code optimization. Automating

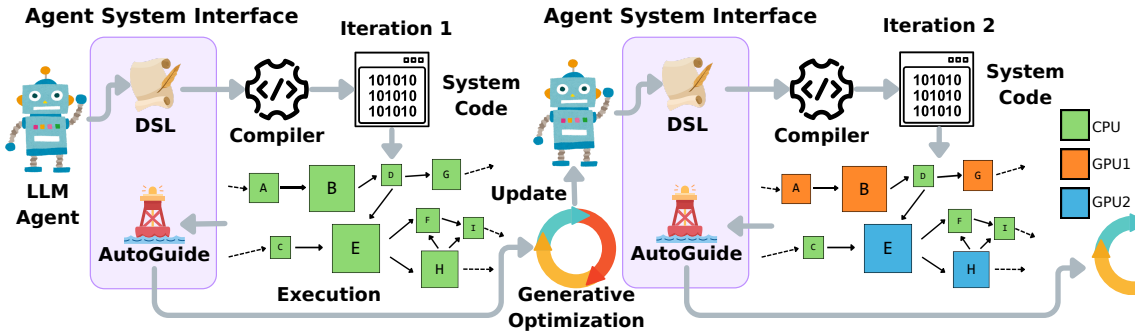


Figure 2.1: Iterative mapper refinement with agent-based generative optimization. The system leverages the Agent-System Interface, which consists of the Domain-Specific Language (DSL) and AutoGuide. The DSL abstracts away the low-level system code, defining a search space for mapping strategies, while AutoGuide interprets execution results into actionable guidance. As iterations progress, the mapper evolves to improve performance.

mapper development would enable scientists to focus on their own domain of expertise while fully utilizing the capabilities of high-performance computing systems.

In this chapter, we introduce a system powered by large language models (LLMs) to **automate both the generation and optimization of mapper code**. The first challenge stems from the *complexity of generating mapper code* due to the original low-level programming system, which exposes the agent to intricate system APIs, coupled with the problem that raw feedback messages from the system are often uninformative to the agent. The second challenge involves *optimizing mapper performance*. Specifically, it consists of (1) defining an appropriate search space and (2) devising efficient methods to find optimal mappers, thereby maximizing parallel program performance.

To address the first challenge, we propose an **Agent-System Interface** (ASI), as shown in Figure 2.1, an abstraction layer between the agent and the system that simplifies code generation and provides more meaningful feedback to the agent. At the core of ASI is a *Domain-Specific Language (DSL)*, a high-level interface that encapsulates all performance-critical decisions required to generate a mapper. The DSL abstracts away the complexity of low-level system code with a compiler. Additionally, the DSL defines a structured search space, enabling systematic exploration of mapping strategies. We also design and implement the *AutoGuide* mechanism to interpret raw execution output into informative and actionable guidance. This mechanism allows the agent to iteratively optimize the mapper by leveraging enriched feedback to update its strategy.

For the second challenge, we adopt the **generative optimization** approach, a recent advance in optimization techniques. Unlike traditional methods such as reinforcement learning [10], which rely solely on scalar rewards, generative optimization can utilize richer forms of feedback, such as error explanations and actionable suggestions expressed in natural language. This agentic optimization workflow has previously proven to be effective across various domains [194, 39, 297, 136, 308]. Our work is the first to apply such technique to the domain of system optimization.

Our experiments demonstrate that mappers optimized by LLM-powered agents not only match but often surpass expert-written mappers, achieving up to 1.34× speedup across nine benchmarks. Since expert-written mappers set the highest standard, surpassing them is a notable accomplishment. At the same time, our method significantly reduces mapper tuning time from days to minutes, making high-performance mapping more accessible to domain scientists. To further highlight the advantage of generative optimization, we compare it against OpenTuner, a reinforcement learning-based autotuning framework. Our generative optimizer finds mappers 11× faster than OpenTuner when both run for 10 iterations and still maintains a 3.8× advantage even when OpenTuner runs for 1000 iterations. Furthermore, ablation studies underscore the necessity of the agent-system interface design in achieving these performance gains. Our contributions are as follows:

1. **Design of an Agent-System Interface:** We introduce an abstraction layer that simplifies mapper code generation and provides guidance to the agent. The Domain-Specific Language (DSL) defines a search space, allowing the agent to explore mapping strategies without dealing with low-level system code. AutoGuide interprets raw execution output into targeted feedback, enabling the agent to refine mapper code more effectively.
2. **Generative Optimization for Systems:** We introduce generative optimization to improve system performance, leveraging richer feedback such as error messages and actionable suggestions in natural language. Unlike reinforcement learning methods like OpenTuner, which rely solely on scalar feedback, our method identifies better mappers in far fewer iterations. With only 10 iterations, it outperforms OpenTuner by 3.8× even after 1000 iterations.
3. **Empirical Evaluation of Performance:** Our agent-based solution achieves up to 1.34× speedup across nine benchmarks, surpassing expert-written mappers while reducing tuning time from days to minutes. We highlight the critical role of the agent-system interface through ablation studies, demonstrating its impact on achieving the performance gains.

2.2 Related Work

Mapping in Parallel Programming Many parallel programming systems allow users to make their own mapping decisions, such as Legion [21], StarPU [12, 11], Chapel [32], HPX [131, 109], Sequoia [76], Ray [190], TaskFlow [120], and Pathways [18]. Several techniques have been proposed to automate mapping, including machine learning models [196, 262], static analysis [206, 211], reinforcement learning [10, 187] and auto-tuning [221]. We use an agent-based approach with LLMs and explore a larger search space for mappers than traditional methods.

Agentic Frameworks Agents powered by Large Language Models (LLMs) play a critical role in decision-making, planning, tool integration, and solving complex problems in dynamic environments [99]. Many agentic frameworks have been developed [305, 281, 148, 114], with uses

spanning domains such as software engineering [102, 301, 130], robotics [134], healthcare [152], education [210], and knowledge engineering [223]. Our work is the first to apply an agentic workflow to iteratively optimize mapper code, improving the performance of parallel programs.

AI for Systems The application of AI to optimize system design has gained significant traction in recent years. Techniques such as deep learning [314, 319, 317] and gradient-boosted trees [79] have been used to predict program execution times for performance optimization. Reinforcement learning methods have addressed challenges in chip floorplanning [186], autotuning [10], auto-vectorization [104], and compiler phase ordering [105]. While previous efforts have predominantly relied on traditional approaches for cost prediction and optimization, our work uses the recent advances in generative optimization to tackle complex system challenges.

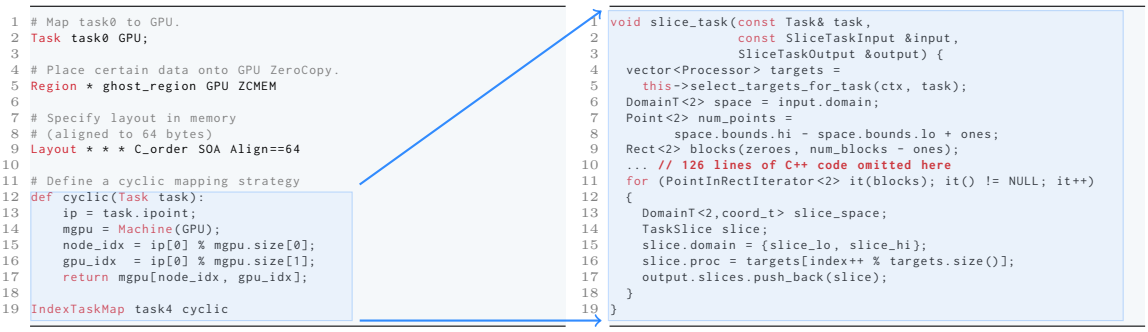
Generative Optimization Recent work has explored the use of LLMs for optimization problems traditionally tackled with numerical methods, including mixed-integer programming [6, 7] and numerical optimization [194]. A key advantage of generative optimization is its ability to iteratively refine solutions using diverse forms of feedback. For example, Trace [39] applies generative optimization to robotic manipulation and game playing, while TextGrad [308] optimizes prompts and molecular designs. While reinforcement learning has been applied to system optimization, the potential of LLM-driven optimization in systems remains unexplored. Our work explores whether generative optimization with richer feedback outperforms traditional methods using scalar rewards in system optimization.

2.3 Problem Definition

Motivation and Challenges The concrete problem we address is the automated generation of high-performance mappers for the Legion parallel programming framework [21]. Mappers dictate task scheduling and data placement. A well-designed mapper can achieve orders-of-magnitude speedup over naive strategies.

However, automating mapper generation is challenging due to two key factors. First, **the complexity of low-level system code**. Implementing a mapper requires writing hundreds of lines of intricate C++ code, demanding expertise in system internals. Second, **the vast search space of mapping strategies**. The search space grows exponentially with the number of tasks and arguments.

Search Space and Performance Impact The search space of mappers involves multiple decisions, each influencing performance. The first key aspect is **processor selection**, which determines whether a task runs on GPUs, CPUs, or the OpenMP runtime. This choice depends on factors such as task size, GPU memory capacity, and kernel launch overhead. For instance, small tasks may prefer CPUs



```
# Map task0 to GPU.
Task task0 GPU;

# Place certain data onto GPU ZeroCopy.
Region * ghost_region GPU ZCMEM

# Specify layout in memory
# (aligned to 64 bytes)
Layout * * * C_order SOA Align==64

# Define a cyclic mapping strategy
def cyclic(Task task):
    ip = task.ipoint;
    mgpu = Machine(GPU);
    node_idx = ip[0] % mgpu.size[0];
    gpu_idx  = ip[0] % mgpu.size[1];
    return mgpu[node_idx, gpu_idx];

IndexTaskMap task4 cyclic
```

(a) An example mapper in Domain-Specific Language (DSL)

```
void slice_task(const Task& task,
                const SliceTaskInput &input,
                SliceTaskOutput &output) {
  vector<Processor> targets =
    this->select_targets_for_task(ctx, task);
  DomainT<2> space = input.domain;
  Point<2> num_points =
        space.bounds.hi - space.bounds.lo + ones;
  Rect<2> blocks(zeroes, num_blocks - ones);
  ... // 126 lines of C++ code omitted here
  for (PointInRectIterator<2> it(blocks); it() != NULL; it++)
  {
    DomainT<2,coord_t> slice_space;
    TaskSlice slice;
    slice.domain = {slice_lo, slice_hi};
    slice.proc = targets[index++ % targets.size()];
    output.slices.push_back(slice);
  }
}
```

(b) Code snippet from a C++ mapper

Figure 2.2: Comparison of a DSL mapper and a C++ mapper. The DSL's declarative, high-level design abstracts away the complexity of low-level C++ code, serving as the core of the Agent-System Interface. The highlighted boxes illustrate how the same functionality, which requires extensive C++ system code, can be expressed concisely in just a few lines in DSL.

due to the overhead of launching GPU kernels, while tasks with large memory footprints may run on CPUs when GPU memory is insufficient.

Another crucial dimension is **memory placement**, which dictates where data is stored. A mapper must decide whether to place data in the GPU's FrameBuffer for fast access, ZeroCopy memory for CPU-GPU sharing, or CPU system memory for more available storage. Each option presents trade-offs between access speed, memory usage, and data transfer overhead.

Additionally, **memory layout** further expands the search space, with decisions on Struct of Arrays (SOA) vs. Array of Structures (AOS), data ordering (Fortran-order vs. C-order), and alignment constraints (e.g., 128-byte alignment) significantly affecting cache efficiency and performance.

Finally, an important idiom in high-performance computing is launching tasks over partitioned data. **Index mapping** determines how data partitions and task executions are distributed across multiple processors. For consistency, we can represent data partitioning as a tensor of data partitions, the machine as a tensor of processors, and tasks operating on the partitioned data as a tensor of tasks. The way data and task indices are mapped to processor indices affects inter-processor communication, a key factor in performance [255, 315].

2.4 Our Approach: Agent-System Interface

2.4.1 Domain-Specific Language Design

A key challenge in automating mapper generation with a coding agent is the complexity of low-level system code, which requires intricate C++ implementations. To address this, we design a high-level **Domain-Specific Language (DSL)** as the core of our **Agent-System Interface** (ASI). The

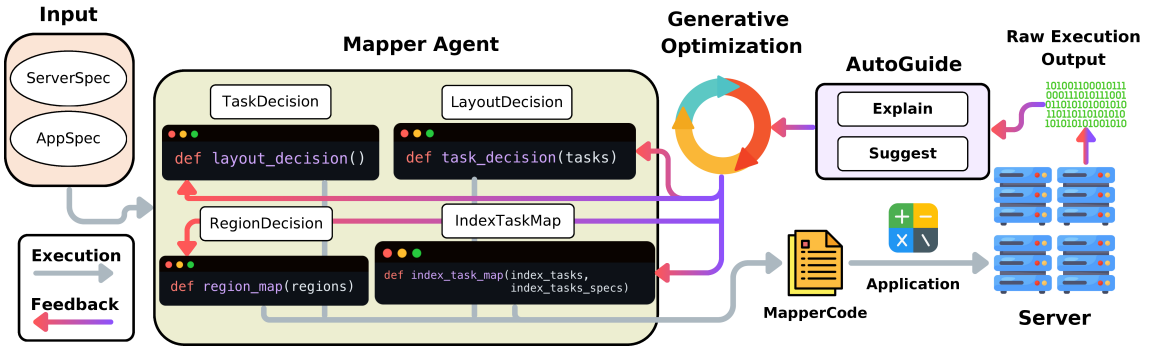


Figure 2.3: Agent Optimization Process. The mapper agent takes server specifications and application-specific information as input, generates mapper code, and executes it alongside the application on the server. Raw execution feedback is enriched using the AutoGuide mechanism, and the mapper is iteratively refined by an LLM optimizer to improve performance.

DSL provides a *structured search space* for mapping strategies while *abstracting away low-level implementation details*. Unlike C++, which demands imperative specifications of mapping policies, our DSL adopts a **declarative design**, allowing users to specify *what* to achieve rather than *how* to implement it. Most critically, the DSL **separates concerns**, enabling multiple aspects of mapping decisions to be expressed *independently rather than being entangled* in low-level system APIs. This design reduces code complexity and naturally provides a search space for the agent to explore. To implement it, we develop a *compiler* that translates DSL into the low-level C++ APIs.

As illustrated in Figure 2.2, the complexity of DSL code is significantly lower than that of C++. Figure 2.2a provides an example of a DSL mapper, highlighting the key features of our DSL. In contrast, Figure 2.2b shows a snippet from a C++ mapper, emphasizing the intricacy of low-level implementation details. Across the benchmarks, using the DSL results in an **average lines of code reduction of** 14×. This substantial reduction makes DSL a more suitable target for LLM code generation, as it abstracts away the complexities inherent in low-level systems. As we will show in Section 2.5.2, LLMs generate DSL code more effectively, despite DSL having no examples in LLM training corpora, whereas C++ is widely represented.

Next, we describe the DSL's design, emphasizing its declarative nature and structured search space. Section 2.3 details the performance impact of each decision.

The `Task` statement (Section 2.3) defines processor selection for each task, choosing between CPU, GPU, or OpenMP. Section 2.3 specifies that instances of `task0` should run on GPUs. This decision is made per task; note that the search space expands exponentially with the number of tasks.

The `Region` statement (Section 2.3) controls memory placement for data arguments. Section 2.3 specifies that all tasks using `ghost_region` should place the data in GPU ZeroCopy memory. Other choices include GPU FrameBuffer memory and CPU System Memory. This decision is made per task and per argument, causing the search space to grow exponentially.

Case	Raw Execution Output	AutoGuide	
		Explain	Suggest
case1	**Compile Error:** Syntax error, unexpected :, expecting {	N/A	There should be no colon : in function definition.
case2	**Compile Error:** IndexTaskMap's function undefined	N/A	Define the IndexTaskMap function first before using it.
case3	**Compile Error:** mgpu not found	N/A	Include `mgpu = Machine(GPU);` in the generated code.
case4	**Execution Error:** Assertion failed: stride does not match expected value.	Memory layout is unexpected.	Adjust the layout constraints or move tasks to different processor types.
case5	**Execution Error:** DGEMM parameter number 8 had an illegal value	Memory layout is unexpected.	Adjust the layout constraint.
case6	**Execution Error:** Slice processor index out of bound	IndexTaskMap statements cause error.	Ensure that the first index of mgpu ends with `% mgpu.size[0]`, and the second element ends with `% mgpu.size[1]`.
case7	**Execution Error:** Assertion 'event.exists()' failed	InstanceLimit statements cause error.	Avoid generating InstanceLimit statements.
case8	**Performance Metric:** Execution time is 0.03s.	N/A	Move more tasks to GPU to reduce execution time.
case9	**Performance Metric:** Achieved throughput = 4877 GFLOPS	N/A	Try using different `IndexTaskMap` or `SingleTaskMap` statements to maximize throughput.

Table 2.1: AutoGuide Feedback Mechanism. The AutoGuide mechanism interprets raw execution output from the runtime system, providing more informative error explanations and suggestions for mapper modifications. It is implemented via keyword matching.

The `Layout` statement (Section 2.3) defines memory layouts. Section 2.3 enforces a `C_order` axis ordering, an `SOA` layout, and a 64-byte memory alignment for all data used by all tasks mapped to all processors. Alternative choices include `F_order`, `AOS`, and various alignment strategies. This is a per-task, per-data, per-processor decision.

The `IndexTaskMap` statement (Section 2.3) controls index mapping using a customized function. Section 2.3 defines the mapping function that establishes the correspondence between two index spaces: the task index space (represented by `task.ipoint`) defined in the application code (e.g., for loops) and the processor space of the distributed machine (represented by `Machine(GPU)`). The DSL

allows users to express arbitrary arithmetic mappings between the two index spaces. This decision applies to each task group launched by parallel for loops.

Our DSL is designed to express a wide range of high-performance mapping strategies, including all of the most important decisions. While there may be cases where certain optimizations are not directly expressible, we have not encountered any. Despite being more constrained than general-purpose C++, the DSL has been proven to be effective: all mappers discovered by our agent that outperform expert-written C++ implementations are expressible within the current DSL.

2.4.2 Generative Optimization via AutoGuide

We formulate mapper generation as an **online optimization problem**. Given a triplet $(\Theta, \omega, \mathcal{T})$, where Θ is a set of possible mappers, ω is an *optimization objective*, and $\mathcal{T}$ is a function that takes a mapper $\theta \in \Theta$ as input, $(f, g) = \mathcal{T}(\theta)$ and returns f, the *feedback* from executing the mapper (i.e., the measured performance after running the application code with the generated mapper), and g, the *process graph* tracing how the mapper was generated. In our setup, mapper performance is deterministic, as we carefully control all sources of randomness in the environment. If the parameter space were numerical, this online optimization problem could be addressed using bandit algorithms [144], reinforcement learning [245], or Bayesian optimization [239], but these methods are less efficient when the parameter search space is large and discrete (i.e., text).

In this online optimization problem, we leverage the DSL to structure the parameter space to improve the efficiency of optimization. Here, θ represents the program code, while ω and f are expressed as text. We adopt **generative optimization**, leveraging LLMs as optimizers given the objective in text form. This emergent optimization behavior has been recently observed and applied across various domains [298, 39, 308, 201].

Optimization Process We present the optimization process in Figure 2.3. The agent takes two inputs: server specifications and application metadata. Server specifications detail the hardware configuration, including the number of CPUs and GPUs per node, as well as the total node count. Application metadata provides information on task names and the associated data arguments accessed by each task. These inputs define the structured search space explored by the agent during optimization. The agent, using the given inputs, generates mapper code that is executed alongside the application code on the server. Raw execution feedback from the runtime is augmented with the AutoGuide mechanism and fed back to the LLM, iteratively refining the agent for improved mapper code generation.

Coding Agent Our mapper agent improves mapping decisions by iteratively generating DSL code. A high-level schema of the mapper agent is shown in Figure 2.3. The mapper agent is implemented as a Python program in the Trace [39] framework, where we decompose the task of generating a

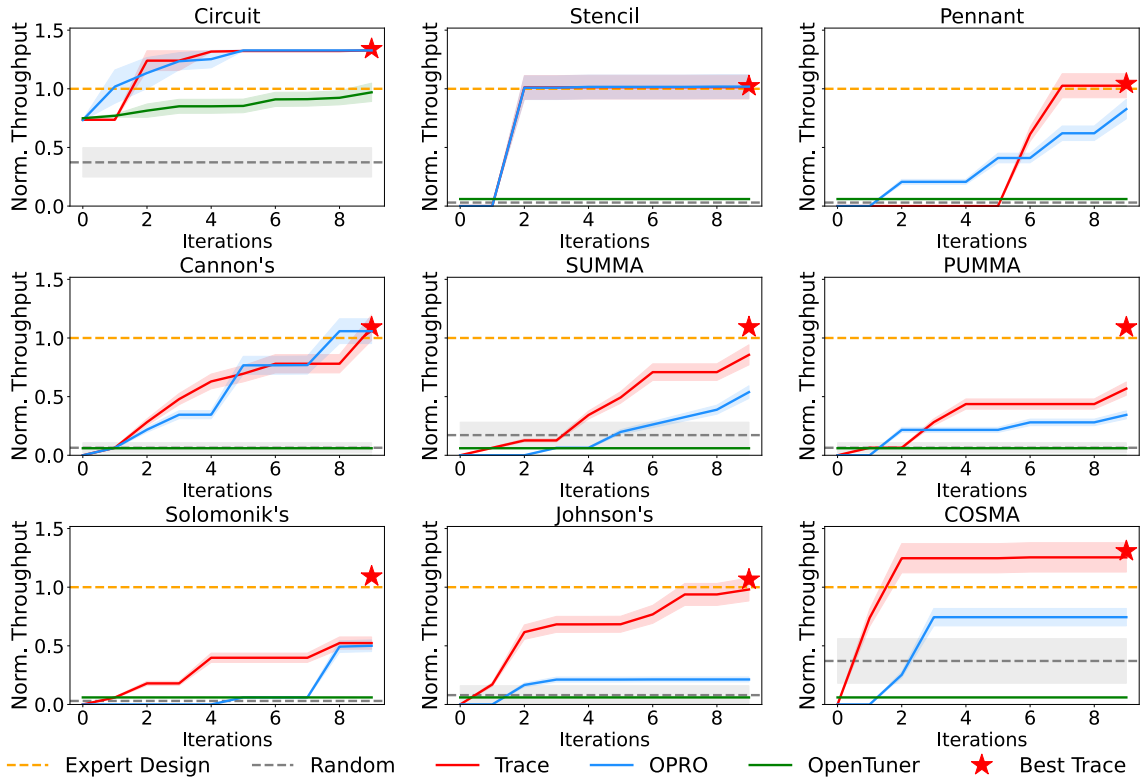


Figure 2.4: Performance Comparison. Normalized throughput for 9 benchmarks, comparing expert mappers, random mappers, the average optimization trajectories of Trace, OPRO, and OpenTuner in 10 iterations across 5 runs, and the best mappers found by Trace.

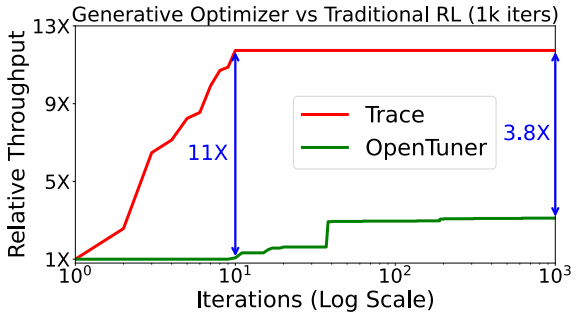


Figure 2.5: Comparison of Trace (generative optimizer) and OpenTuner (traditional RL) over 1K iterations (averaged across all 9 benchmarks).

monolithic mapper into *independent code segments*. This decomposition allows the agent to decide what code to generate for each segment separately. This approach is effective because our *DSL design eliminates unnecessary dependencies* between mapping decisions. Our modularization strategy aligns with least-to-most prompting [321].

AutoGuide The AutoGuide feedback mechanism is designed based on three key motivations: (1) generative optimization benefits from natural language feedback rather than relying solely on scalar values, (2) raw execution output from the runtime system is often too uninformative to effectively guide the agent's decisions, and (3) domain heuristics known to systems researchers can be naturally expressed in language (e.g., most tasks run faster on GPUs than CPUs). To address these needs, AutoGuide helps the agent by **explaining** opaque error messages and **suggesting** mapper modifications. As shown in Table 2.1, it interprets uninformative execution output into actionable insights. The implementation relies on keyword matching over the raw execution output. An ablation study in Section 2.5.3 demonstrates its effectiveness in our experiments.

2.5 Evaluation

Experiments are conducted on one node with two Intel 10-core E5-2640 v4 CPUs, 256G main memory, and four NVIDIA Tesla P100 GPUs. We use gpt-4o-2024-08-06.

2.5.1 Speedup of Application Performance

Benchmarks Our evaluation utilizes a suite of 9 benchmarks, including 3 scientific computing workloads and 6 well-known matrix multiplication algorithms. *Circuit* is a simulation benchmark that models electrical circuit behavior by simulating currents and voltages across interconnected nodes and wires [21]. *Stencil* simulates a 2D grid where each point's value is updated based on a stencil pattern determined by its neighbors [257]. *Pennant* models unstructured mesh Lagrangian staggered-grid hydrodynamics, commonly used for simulating compressible flow [82]. The remaining six benchmarks – *Cannon's*, *SUMMA*, *PUMMA*, *Johnson's*, *Solomonik's*, and *COSMA* – are well-known parallel matrix multiplication algorithms [29, 256, 40, 4, 240, 140]. Parallel matrix multiplication remains an active research topic due to its central role in high-performance computing and scientific simulations [291]. Furthermore, improving matrix multiplication performance has a broad impact, as it accelerates numerous downstream machine learning workloads [125, 318]. This benchmark suite provides both depth with its representative matrix multiplication algorithms and variety with its range of scientific computing workloads.

In this experiment, we evaluate the performance of the mappers with the following baselines.

Expert-Written Mappers. These mappers are manually developed by domain scientists who spend years mastering computational science. Writing mappers in parallel programming frameworks is another challenge, and tuning them for specific applications can take days.

Randomly Generated Mappers. These mappers were randomly generated with 10 different random seeds, sampling from the entire search space of each application. We report the average performance.

Agent-Optimized Mappers. Using Trace [39], we evaluated the **Trace** and **OPRO** [297] search

Code Generation Target	Mapping Strategy										Success Rate
	1	2	3	4	5	6	7	8	9	10	
C++ (single trial)	✗	–	–	✗	–	–	✗	✗	–	–	0%
DSL (single trial)	✓	✓	✓	✓	✓	–	✓	✓	✓	–	80%
C++ (iterative refine)	✗	–	–	✗	✗	✗	✗	✗	✗	✗	0%
DSL (iterative refine)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	100%

Table 2.2: Code Generation Success Rates. Success rates for generating code across 10 mapping strategies described in natural language. The test evaluates whether the generated code compiles and passes execution tests. Generating DSL code significantly outperforms generating C++ for both settings. Symbols indicate results: – fails to compile, ✗ compiles but fails the test, and ✓ passes the test.

algorithms, running 10 iterations per application. To account for stochastic output, we repeated the process 5 times and report the average. The best mapper from Trace across runs is also reported.

OpenTuner Mappers. OpenTuner [10] is a program autotuning framework that uses reinforcement learning to optimize performance based on scalar feedback. We provided execution time as feedback, with a high penalty for failures.

Results In Figure 2.4, we use normalized throughput as our performance metric, where higher values indicate better performance. The throughput is normalized relative to the expert-written mappers, providing a clear baseline for comparison. Our focus is on measuring end-to-end performance, which includes both the correctness and efficiency of the generated mappers. If the generated code has any syntax or runtime issues, its throughput is recorded as 0. We report the best mappers found by Trace, and the average optimization trajectories of Trace, OPRO and OpenTuner over 10 iterations across 5 runs.

All the best mappers found by Trace can match or surpass the expert-written mappers, underscoring the effectiveness of agent-based generative optimizer. In our context, reporting the best-performing mapper is appropriate. Mapper optimization is an offline process, and in practice, it is standard to run the optimizer multiple times and deploy the best result. Once identified, the mapper can be reused across repeated executions on the same application, input, and hardware, incurring no further search cost.

Random mappers consistently exhibit low performance across all applications, emphasizing the critical role of mapping decisions. For each application, we generate 10 random mappers by sampling from the full DSL-defined search space, totaling 90 mappers across 9 applications. Among them, 74 (82.2%) raise runtime errors due to invalid mapping decisions. The runtime system enforces correctness by rejecting such mappers during execution, resulting in a throughput of 0.

When comparing optimization trajectories, Trace performs similarly to OPRO, and significantly outperforms OpenTuner. To further compare the agent-based optimizer with traditional reinforcement

learning, we extended OpenTuner's optimization iterations from 10 to 1000, as shown in Figure 2.5, where the x-axis is the log-scale of iterations and the y-axis represents relative throughput (averaged across all 9 benchmarks). Notably, Trace achieves a 3.8× speedup over OpenTuner even when OpenTuner is run for 1000 iterations. When both are limited to 10 iterations, Trace outperforms OpenTuner by 11×, demonstrating its ability to quickly identify high-performance mappings. **This highlights the superiority of Trace (generative optimizer) over OpenTuner (traditional reinforcement learning).** Moreover, Trace completes the entire optimization process in just 10 minutes per application, **reducing mapper development time from days to minutes**.

To offer a more comprehensive view of performance variations, we present additional statistics, including the mean, standard deviation, worst, median, and best normalized throughput for both our method and OpenTuner. These statistics are derived from five runs for each benchmark.

Case Analysis The largest performance gain achieved by Trace over the expert mapper is observed in Circuit, with a speedup of 1.34×. This improvement is primarily due to *memory placement*: the best mapper allocates two data collections to GPU FrameBuffer memory, while the expert mapper places them in GPU ZeroCopy memory. Despite a slight increase in inter-GPU communication costs, Trace reduces task execution time due to faster memory access, resulting in higher overall performance. For matrix-multiplication algorithms, the greatest speedup is seen in COSMA, with Trace achieving a 1.31× speedup over the expert mapper. This is attributed to Trace's more efficient index mapping functions, which *reduce inter-GPU communication* by better distributing partitioned submatrices across GPUs.

2.5.2 Ablation Study of DSL for Code Generation

In Section 2.5.1, we demonstrate the overall effectiveness of our approach. Here, we conduct an ablation study on the DSL, the core of the Agent-System Interface. Since successful generation is the foundation of optimization, this subsection focuses on **how well the DSL helps LLMs *generate* syntactically and semantically correct mappers compared to C++**, rather than directly *optimizing* performance.

Experiment Setup We designed 10 mapping strategies, described in natural language, to evaluate whether LLMs can generate correct code in both the DSL and the original low-level C++. To ensure a fair comparison, identical prompt materials (documentation, examples, and starting code) were provided for both the DSL and C++. Success rates are measured based on whether the generated code passes predefined test cases, with results reported for single trials and iterative refinement, where the LLM is allowed up to 10 iterations to improve the code using compiler feedback. The evaluation is conducted with the DSPy [136] framework.

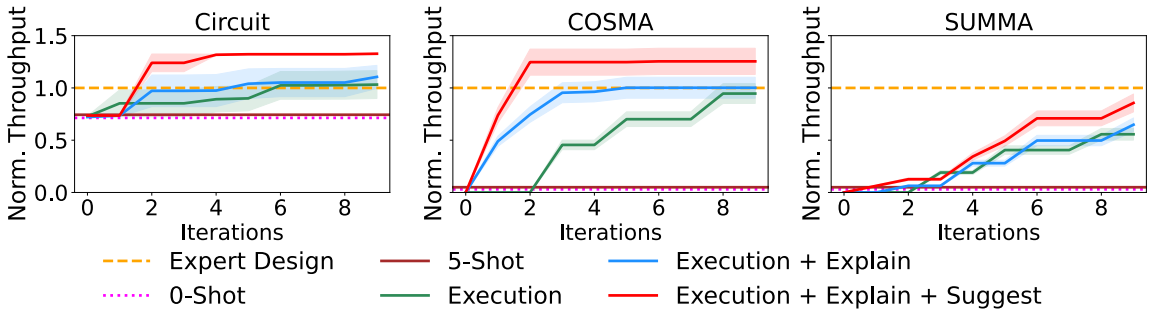


Figure 2.6: Comparison of different feedback designs. 0-Shot and 5-Shot are baselines. Execution provides only the raw execution output as feedback. Explain provides additional explanations of execution errors. Suggest offers mapper modification suggestions. All feedback is automatically generated.

Results Table 2.2 shows that **DSL achieves significantly higher generation success rates than C++** in both the single-trial and iterative refinement settings. This demonstrates the effectiveness of DSL's design in abstracting system complexity and providing a high-level interface that enables LLMs to tackle complex system challenges in code generation. Incorporating iterative refinement with compiler feedback further improves success rates, resolving four compilation errors in C++ and two in the DSL. However, the gap between DSL mappers and C++ mappers remains substantial. Notably, these results are striking given that the DSL is a low-resource language with no pre-training or fine-tuning data, while C++ code is widely present in LLM training corpora.

Analysis LLMs perform better with the DSL for two reasons. First, the **semantic gap** between natural language and code is smaller with the DSL than with C++. For example, writing a mapper to "align all data to 64 bytes in memory and use Fortran ordering" requires one line `Layout * * * Align==64 F_order` in the DSL because of its *declarative design*. In contrast, the C++ mapping API requires a sequence of operations to enforce alignment and ordering, which widens the semantic gap. Second, the DSL reduces the **amount of code**. As discussed before, LLMs achieve an average reduction of 14× in lines of code, simplifying code generation. These results underscore the importance of a high-level agent-system interface.

2.5.3 Ablation Study of the AutoGuide Feedback

The AutoGuide mechanism provides enriched feedback to the agentic optimizer. We compare with alternative feedback designs.

Experiment Setup We compare the following baselines. **0-shot** and **5-shot** have no feedback, allowing the LLM to generate once with either 0 or 5 examples provided. **Execution** only provides raw execution feedback, **Explain** offers additional explanations for execution errors, and **Suggest**

offers mapper modification suggestions. The Trace trajectory shown in Figure 2.4 uses the full AutoGuide mode with all **Execution+Explain+Suggest**. As an ablation study, we evaluate 3 benchmarks.

Results and Analysis Figure 2.6 demonstrates that the **full feedback mechanism consistently outperforms** all reduced feedback variants. The 0-shot and 5-shot results perform the worst, underscoring the importance of feedback-based iterative refinement. This highlights the value of an agentic workflow, showing that performance improvements are not solely driven by prompting the LLM but are a direct result of the iterative refinement in the workflow design.

2.6 Conclusion

In this chapter, we introduced a system that leverages LLMs to automate mapper generation and optimization. The Agent-System Interface (ASI) simplifies code generation with a Domain-Specific Language (DSL), which abstracts away the low-level complexity of system code, and enriches execution feedback through AutoGuide, which interprets raw execution output into actionable guidance. We adopted generative optimization, allowing LLMs to refine mappers using rich textual feedback beyond scalar metrics. Unlike RL-based methods like OpenTuner, which rely on numerical rewards, our approach incorporates error explanations and targeted suggestions, accelerating search efficiency. Experiments show that agent-generated mappers outperform expert-written ones, achieving up to 1.34× speedup across nine benchmarks. Our method, running only 10 iterations, maintains a 3.8× advantage over OpenTuner even after 1000 iterations. By reducing mapper development time from days to minutes, our approach benefits computational scientists and demonstrates the effectiveness of generative optimization in system design.

Chapter 3

Optimizing Assembly Programs with Language Models

3.1 Introduction

Superoptimization is the task of transforming a program into a faster one while preserving its input-output behavior. In this chapter, we investigate whether large language models (LLMs) can perform superoptimization by generating assembly code that surpasses the performance of compiler outputs.

Decades of research have tackled the problem of code optimization, giving rise to two main approaches. The first develops better algorithms for rule-based transformations in compilers [275]. However, given the vast space of possible transformations, compiler-optimized code is not guaranteed to be optimal and often leaves performance untapped [30, 250]. The second, superoptimization, develops search algorithms that directly explore the space of all possible programs, aiming to discover the correct variant with the best performance rather than relying on a fixed set of transformation rules [216].

Superoptimization is an aggressive form of program optimization that can outperform compiler-optimized code, yet existing literature has focused on very short, straight-line assembly programs without loops. Prior work has primarily relied on CPU-based search heuristics, which fail to scale to larger programs [216, 205, 137]; available datasets include at most 15 lines of straight-line assembly [138].

In this chapter, we explore the use of LLMs as a superoptimizer to improve the performance of assembly code. In contrast to most prior work on code generation from natural language [34, 13, 110, 323], we tackle a fundamentally different and more technically demanding task: improving assembly code that has already been optimized by the industry-standard compiler at its highest

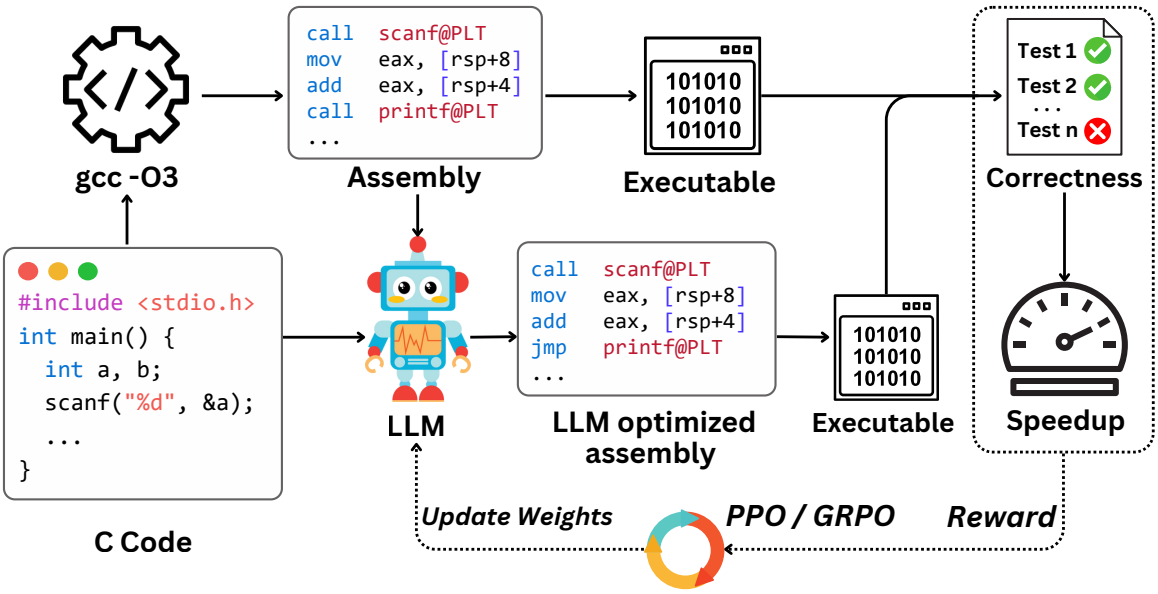


Figure 3.1: Overview of the assembly code optimization task. Given a C program and its baseline assembly from gcc -O3, an LLM is fine-tuned with PPO or GRPO to generate improved assembly. The reward function reflects correctness and performance based on test execution.

optimization level (gcc -O3). Compilers have been refined over decades of expert-driven development, and surpassing them remains a central challenge in programming languages, as compilers form the foundation of all software.

Unlike high-level programming languages (e.g., Python or C), large-scale, high-quality assembly datasets are scarce. As the first study in this direction, we construct a dataset of 8,072 assembly programs. Each instance includes input–output test cases and baseline assembly generated by the compiler at its highest optimization level (gcc -O3), which serves as the starting point for further optimization. In contrast, the datasets commonly used in the superoptimization community [263, 216, 138] are either extremely limited in size, containing only 25 programs, or consist of toy examples with 2 to 15 instructions without loops. Our dataset is substantially larger, with assembly programs averaging 130 lines and including loops. Our test suites achieve 96.2% line and 87.3% branch coverage, demonstrating strong test quality. Our dataset represents a substantial step forward in scale for evaluating superoptimization techniques.

Beyond evaluating existing models, we also apply reinforcement learning (RL) for fine-tuning to further enhance their capabilities. We use widely adopted algorithms, including Proximal Policy Optimization (PPO) and Group Relative Policy Optimization (GRPO), to train an LLM with a reward function that integrates both correctness and performance speedup. Prior work on LLM-based performance optimization has explored alternative methodologies such as supervised fine-tuning [233], chain-of-thought prompting [169], agent-based frameworks [265, 266], and preference learning [64].

Our approach optimizes speedup explicitly in the reward function, making reinforcement learning well-suited to superoptimization. To our knowledge, this is among the first applications of reward-based RL to LLMs for code performance optimization, with correctness and speedup jointly encoded in the objective.

We evaluate 23 LLMs on this task and find that the best-performing model, Claude-opus-4, achieves a 51.5% test-passing rate and an average speedup of 1.43× over the compiler-optimized baseline (gcc -O3). Our reinforcement learning approach is highly effective: starting from the base model Qwen2.5-Coder-7B-Instruct, which achieves 61.4% correctness and a modest 1.10× speedup, the fine-tuned model SuperCoder attains 95.0% correctness and 1.46× average speedup, with further improvement enabled by Best-of-N sampling and iterative refinement.

In summary, our contributions are as follows:

- We are the first to introduce superoptimization as a task for LLMs, a technically demanding challenge that aims to improve assembly code already optimized by industry-standard compilers.
- We construct the first large-scale dataset of 8,072 assembly programs, averaging 130 lines. This far surpasses prior loop-free datasets under 15 lines and marks a substantial step forward in scale and realism for evaluating superoptimization techniques.
- We evaluate 23 LLMs on the benchmark and show that RL-based training substantially improves performance: fine-tuning Qwen2.5-Coder-7B-Instruct (61.4% correctness, 1.10× speedup) results in SuperCoder with 95.0% correctness and 1.46× speedup, with further gains enabled by Best-of-N sampling and iterative refinement.

3.2 Related Work

Large Language Models for Code. Benchmarks for evaluating large language models (LLMs) on code generation from natural language specifications have received increasing attention. Notable examples include HumanEval [34], MBPP [13], APPS [110], and more recent efforts [167, 149, 284, 323]. In parallel, many models have been developed to enhance code generation capabilities, such as Codex [34], AlphaCode [157], CodeGen [195], InCoder [87], StarCoder [153], DeepSeek-Coder [98], Code Llama [213], and others [121, 270]. Beyond code generation, LLMs have been applied to real-world software engineering tasks including automated program repair [287, 286], software testing [285, 59], bug localization [298], transpilation [303, 23], equivalence checking [264], and synthesis [267]. SWE-bench [128] integrates these tasks into a benchmark for resolving real GitHub issues. Building on this, SWE-agent [301] and subsequent works [283, 273] employ an agent-based framework that leverages LLMs to improve the issue resolution process.

Recent work has also explored LLMs for improving program performance. CodeRosetta [249]

targets automatic parallelization, such as translating C++ to CUDA. Other efforts focus on optimizing Python code for efficiency [64, 169] or enabling self-adaptation [116], and improving C++ performance [233]. Of particular relevance are approaches to low-level code optimization [265, 197]. The LLM Compiler foundation models [51, 52] are primarily designed for code size reduction and binary disassembly, whereas our work focuses on optimizing assembly code for performance. LLM-Vectorizer [247] offers a formally verified solution for auto-vectorization, a specific compiler pass. In contrast, our work does not restrict the optimization type and uses test-case validation.

Learning-Based Code Optimization. The space of code optimization is vast, and many approaches have leveraged machine learning to improve program performance. A classic challenge in compilers is the phase-ordering problem, where performance depends heavily on the sequence of optimization passes. AutoPhase [105] uses deep reinforcement learning to tackle this, while Coreset [158] employs graph neural networks (GNNs) to guide optimization decisions. Modern compilers apply extensive rewrite rules but offer no guarantee of optimality. Superoptimization seeks the most efficient program among all semantically equivalent variants of the compiler output. Traditional methods use stochastic search, such as MCMC sampling [216], with follow-up work improving scalability [205, 27] and extending to broader domains [229, 43]. These rely on formal verification for correctness, restricting them to small, loop-free programs. In contrast, our approach uses test-based validation, enabling optimization of general programs with loops. With the rise of deep learning, substantial attention has turned to optimizing GPU kernel code. AutoTVM [37] pioneered statistical cost model-based search for CUDA code optimization, followed by methods such as Ansor [314], AMOS [317], and others [222, 313, 280].

More recently, LLMs have been explored as code optimizers [233, 93, 265, 269, 266], with growing interest in reinforcement learning that guides generation through reward signals [63, 273]. Rewards are often derived from unit-test correctness [146, 232, 165] or binary preference signals [166, 64]. To our knowledge, this is among the first works to apply reinforcement learning to optimize code performance with LLMs, with concurrent efforts exploring CUDA kernel optimization [156, 20].

3.3 Methodology

3.3.1 Task Definition

Let C be a program written in a high-level language such as C. A modern compiler like gcc can compile C into an x86-64 assembly program $P = gcc(C)$, which can then be further assembled into an executable binary. The assembly program P serves as an intermediate representation that exposes low-level optimization opportunities, making it suitable for aggressive performance improvement. We assume the semantics-preserving nature of the compilation process, i.e., $[\![C]\!] = [\![P]\!]$, so that the behavior of the assembly program P is identical to that of the source program C.

In theory, the goal is to produce a program P' that is functionally equivalent to P across the entire input space $\mathcal{X}$, i.e., $P(x) = P'(x)$ for all $x \in \mathcal{X}$. Since verifying this property is undecidable in general, we approximate equivalence using a finite test set $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^{n}$, where each input-output pair (x_i, y_i) captures the expected behavior of C.

We say that an assembly program P' is *valid* if it can be successfully assembled and linked into an executable binary. Let $\texttt{valid}(P') \in \{\texttt{True}, \texttt{False}\}$ denote this property. Based on all that we said above, we define the set of *correct* programs as:

$$\mathcal{S}(P) = \{P' \mid \texttt{valid}(P') \ \wedge \ \forall (x_i, y_i) \in \mathcal{T}, \ P'(x_i) = y_i\}.$$

Performance and Speedup. Let $t(P)$ denote the execution time of P on the test set $\mathcal{T}$, and let $t(P')$ be the corresponding execution time for P'. The speedup of P' relative to P is defined as follows. If the LLM-generated program is invalid or slower, we fall back to the baseline and assign a speedup of 1.

$$\text{Speedup}(P') = \begin{cases} \frac{t(P)}{t(P')} & \text{if } P' \in \mathcal{S}(P) \text{ and } t(P') < t(P), \\ 1 & \text{otherwise.} \end{cases}$$

Optimization Objective. The objective is to generate a candidate program P' that maximizes $\text{Speedup}(P')$. Only programs in $\mathcal{S}(P)$ are eligible for speedup; any candidate that fails to compile into a binary or produces incorrect outputs is assigned a default speedup of 1. This reflects a practical fallback: when the generated program is invalid, the system can revert to the baseline P, compiled with gcc -O3, which defines the $1\times$ reference performance. Although $\mathcal{S}(P)$ captures the correctness criteria, we do not restrict the LLM to generate only valid programs. Instead, the model produces arbitrary assembly code, and correctness is validated post hoc via compilation and test execution. We train an LLM using reinforcement learning (see Section 3.3.3) to generate candidates that both satisfy correctness and achieve performance improvements.

3.3.2 Dataset Construction

We construct our dataset using C programs from CodeNet [208], a large-scale corpus of competitive programming submissions. CodeNet is a well-established and widely used benchmark in the AI-for-code community [157, 233]. Each dataset instance is a tuple $(C, P, \mathcal{T})$, where C is the original C source code, $P = gcc(C)$ is the corresponding x86-64 assembly generated by compiling C with gcc at the -O3 optimization level, and $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^{n}$ is the test set. Since not all CodeNet problems include test inputs, we adopt those provided by prior work [157] to define x_i, but discard their output labels. Instead, we regenerate each output y_i by executing the original submission on input x_i, as many CodeNet programs are not accepted solutions, and even accepted ones do not reliably pass all

test cases.

Given the scale of CodeNet, which contains over 8 million C and C++ submissions, we sample a subset for this study. To focus on performance-critical cases, we sample programs that exhibit the highest relative speedup from gcc -O0 (no optimization) to gcc -O3 (maximum optimization). Such a strategy serves two purposes: (1) it favors programs with complex logic that lead to suboptimal performance under -O0 and can be effectively optimized by -O3, and (2) it creates a more challenging setting by starting from code that has already benefited from aggressive compiler optimizations. The final dataset consists of 7,872 training programs and 200 held-out evaluation programs, with additional statistics provided in Section 3.4.

3.3.3 Reinforcement Learning

We conceptualize our task as a standard contextual multi-armed bandit problem [175], defined by a context space $\mathcal{S}$, an action space $\mathcal{A}$, and a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Each context $s \in \mathcal{S}$ represents a problem instance, comprising the source program C, its baseline assembly P, and the associated test cases $\mathcal{T}$. An action $a \in \mathcal{A}$ corresponds to generating a candidate assembly program $\tilde{P}$. The reward function $r(s, a)$ evaluates the quality of the generated program based on correctness and performance. We will describe different designs of the reward function later. A policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ maps a context s to a probability distribution over actions and samples an action $a \in \mathcal{A}$ stochastically. Given a distribution μ over problem instances, the expected performance of a policy π under reward function r is expressed as $\mathbb{E}_{s\sim\mu, a\sim\pi(\cdot|s)}\left[r(s, a)\right]$. The objective is to find a policy that maximizes this expected reward.

Optimization with PPO and GRPO. We train the policy using two policy-gradient algorithms: *Proximal Policy Optimization* (PPO) [218] and *Group Relative Policy Optimization* (GRPO) [225]. PPO stabilizes training by constraining each update to remain close to the previous policy. It maximizes a clipped surrogate objective of the form $\mathbb{E}_{s,a}\left[\min\left(\rho(\theta)\hat{A},\ \text{clip}(\rho(\theta), 1-\epsilon, 1+\epsilon)\,\hat{A}\right)\right]$, where $\rho(\theta) = \pi_\theta(a \mid s)/\pi_{\theta_{\text{old}}}(a \mid s)$, $\hat{A}$ is the estimated advantage, and ϵ controls the clipping range. GRPO, in contrast, compares rewards among a group of sampled outputs and assigns a higher likelihood to relatively stronger ones, effectively normalizing advantages without requiring a value function. In both algorithms, rewards are based on the correctness and execution time of the generated program, eliminating the need for a separate reward model.

Reward Function Design. As defined in our contextual bandit setup, the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ assigns a scalar score to each (context, action) pair. Each context $s \in \mathcal{S}$ consists of the source program C, the baseline assembly P, and a test set $\mathcal{T} = \{(x_i, y_i)\}_{i=1}^{n}$. An action $a \in \mathcal{A}$ corresponds to a generation procedure that produces a candidate assembly program $\tilde{P} = \mathsf{gen}(a)$.

We define two auxiliary metrics for computing reward:

$$\text{pass}(s, a) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \mathbf{1}[\tilde{P}(x) = y],$$

$$\text{speedup}(s, a) = t(P)/t(\tilde{P}),$$

which denote the fraction of test cases passed and the speedup of the generated program $\tilde{P}$ relative to the baseline P. We use the following reward function during training:

$$r(s, a) = \begin{cases} 0, & \text{if } \text{pass}(s, a) < 1, \\ \text{speedup}(s, a), & \text{if } \text{pass}(s, a) = 1. \end{cases}$$

If a generated program fails to compile or does not pass all tests, its reward is set to 0, with no partial credit for partial correctness. Only when the code compiles and passes all tests is a positive reward assigned, equal to the achieved speedup.

3.3.4 Best-of-N Sampling, Supervised Fine-Tuning, and Iterative Refinement

Best-of-N Sampling. Generating multiple candidate programs and selecting the strongest one is a well-established strategy for improving code generation quality [157, 68]. In our setting, the best candidate refers to the program that is correct while achieving the fastest execution time. Best-of-N sampling is an inference-time technique that can boost performance, but it incurs additional cost because each candidate is tested at runtime.

Supervised Fine-Tuning. To obtain training targets for supervised fine-tuning, we require reference solutions to the superoptimization task. However, superoptimization is inherently open-ended: beyond the compiler baseline, there is no unique ground-truth program, and multiple distinct solutions may exist. We apply best-of-8 sampling with the base model and treat the highest-quality candidate for each instance as the ground truth. We then fine-tune the model using LoRA [115].

Iterative Refinement. Iterative refinement is a complementary inference-time technique that can be applied to any model to further improve its outputs. After each trial, we feed back the model's previous attempt: if the generated program fails to compile or fails any test cases, we include the corresponding compiler errors or test failures in the next prompt; if the model produces a correct program, we also include that successful attempt as part of the prompt.

Split	# Prog.	Avg. Tests	Avg. LOC	
			C	Assembly
Training	7,872	8.86	22.3	130.3
Evaluation	200	8.92	21.9	133.3

Table 3.1: Dataset statistics across training and evaluation splits. LOC = lines of code.

3.4 Experimental Setup

Dataset. We describe our dataset construction approach in Section 3.3.2. Each instance consists of a C source program C, the corresponding gcc -O3 compiled assembly P, and a set of test cases $\mathcal{T}$ for correctness evaluation. The final dataset contains 7,872 training programs and 200 evaluation programs, with average program lengths and test case counts summarized in Section 3.4, and additional analysis below.

Test Coverage. Our dataset includes test cases for every program. Rather than relying directly on the original submissions, we re-run each program on its inputs to generate correct outputs, thereby fixing errors in prior datasets. The resulting test suites of our evaluation dataset achieve an average of 96.2% line coverage and 87.3% branch coverage, demonstrating high test quality.

Speedup by Compilers. We quantify compiler optimizations by comparing gcc -O0 with gcc -O3 on the evaluation dataset and observe a mean speedup of 2.65×. This demonstrates the substantial effect of compiler optimizations and confirms that performance improvements in our dataset are measurable. Building on this baseline, we investigate whether LLMs can further enhance performance beyond the 2.65× speedup provided by the compiler.

Prompts. For each instance, we construct a prompt that includes the original C program along with the generated assembly using gcc -O3. Test cases are withheld from the model.

Metrics. We evaluate each model using both correctness and performance metrics. *Compile pass* is the percentage of problems for which the generated assembly compiles to binary executable successfully, while *test pass* is the percentage of problems where the compiled code passes all test cases. For a given problem, any single failed test case is considered a failure for the test pass metric. Both metrics are computed across the entire validation set. For performance, we measure the relative speedup over the gcc -O3 baseline. As defined in Section 3.3.1, we assign a default speedup of 1× to any candidate that fails to compile, fails any test case, or is slower than the baseline. This reflects the practical setting where a system can fall back to the gcc -O3 output, resulting in no performance

gain. We report the *25th*, *50th* (median), and *75th* percentiles of speedup to capture distributional behavior, along with the *average speedup* over the entire evaluation set.

Models. We evaluate 23 state-of-the-art language models spanning a diverse range of architectures. Our benchmark includes frontier proprietary models such as gpt-4o [3], o4-mini, gemini-2.0-flash-001 [248], and claude-3.7-sonnet, as well as open-source families such as Llama [252], DeepSeek [160], and Qwen [121]. In addition, we include models distilled from DeepSeek-R1 [97] based on Qwen and Llama. Finally, we evaluate recent compiler-oriented foundation models [51, 52], pretrained on assembly code and derived from Code Llama, with a design focus on compiler tasks (listed as llm-compiler in Table 3.2). All open-source models are instruction-tuned.

Performance Measurement. To ensure an accurate performance evaluation, we use hyperfine [1], a benchmarking tool that reduces measurement noise by performing warmup runs followed by repeated timed executions. For each program's execution, we discard the first three runs and report the average runtime over the next ten runs.

Implementation. We implement our customized reinforcement learning reward functions within the VERL framework [230], which enables fine-tuning of LLMs using PPO and GRPO. As part of this setup, we build a task-specific environment that handles program compilation, test execution, and runtime measurement, as detailed in Section 3.3.3. This environment provides the model with direct scalar feedback based on both functional correctness and execution performance.

Training Configurations. Among all evaluated models (see Table 3.2), we select Qwen2.5-Coder-7B-Instruct for training due to its strong correctness results and substantial room for performance improvement, while intentionally avoiding compiler-specific foundation models to preserve generality. Training is performed on a single node with four A100 GPUs.

3.5 Results

3.5.1 Evaluation of Different Models

Main Results. Table 3.2 reports results across evaluated models. Most perform poorly on this task, with only a few demonstrating effectiveness as superoptimizers. Most models struggle to generate performant assembly: the majority yield only 1.00× speedup, with low compile and test pass rates. Among all models, claude-opus-4 and claude-sonnet-4 perform best, achieving average speedups of 1.43× and 1.30×, respectively. Compiler foundation models (prefixed with llm-compiler-) are pretrained on assembly code and compiler IRs. Among them, llm-compiler-13b achieves a notable 1.34× speedup, whereas the fine-tuned variants (-ftd) perform poorly, likely because they were

Model	Compile Pass	Test Pass	Speedup Percentiles 25th	50th	75th	Average Speedup
DS-R1-Distill-Qwen-1.5B	0.0%	0.0%	1.00×	1.00×	1.00×	1.00×
DeepSeek-R1	0.0%	0.0%	1.00×	1.00×	1.00×	1.00×
DS-R1-Distill-Llama-70B	5.5%	0.0%	1.00×	1.00×	1.00×	1.00×
DS-R1-Distill-Qwen-14B	11.5%	0.5%	1.00×	1.00×	1.00×	1.00×
gpt-4o-mini	44.5%	1.0%	1.00×	1.00×	1.00×	1.00×
Llama-4-Maverick-17B	77.5%	7.0%	1.00×	1.00×	1.00×	1.02×
Llama-3.2-11B	84.0%	21.0%	1.00×	1.00×	1.00×	1.02×
gpt-4o	81.0%	5.0%	1.00×	1.00×	1.00×	1.02×
Llama-4-Scout-17B	68.5%	5.5%	1.00×	1.00×	1.00×	1.02×
o4-mini	25.0%	4.5%	1.00×	1.00×	1.00×	1.02×
gemini-2.0-flash-001	57.5%	4.0%	1.00×	1.00×	1.00×	1.03×
Qwen2.5-72B	59.5%	7.5%	1.00×	1.00×	1.00×	1.03×
Llama-3.2-90B	82.5%	15.0%	1.00×	1.00×	1.00×	1.05×
Qwen2.5-Coder-7B	77.9%	61.4%	1.00×	1.00×	1.00×	1.10×
gpt-5	78.5%	6.0%	1.00×	1.00×	1.00×	1.13×
DeepSeek-V3	94.0%	43.0%	1.00×	1.00×	1.40×	1.21×
claude-3.7-sonnet	94.5%	58.5%	1.00×	1.10×	1.45×	1.22×
claude-sonnet-4	87.0%	37.0%	1.00×	1.00×	1.95×	1.30×
claude-opus-4	90.0%	51.5%	1.00×	1.58×	2.03×	1.43×
llm-compiler-7b-ftd	2.0%	2.0%	1.00×	1.00×	1.00×	1.00×
llm-compiler-13b-ftd	2.5%	2.0%	1.00×	1.00×	1.00×	1.01×
llm-compiler-7b	55.0%	54.0%	1.00×	1.00×	1.00×	1.09×
llm-compiler-13b	60.5%	59.5%	1.00×	1.27×	1.63×	1.34×

Table 3.2: Comparison of LLMs on our assembly optimization benchmark. We report compilation success rate, test pass rate, and average speedup over the gcc -O3 baseline. Speedup is averaged across all test inputs, with each input evaluated over ten runs.

adapted for different tasks such as disassembling x86-64 into LLVM IR. These results suggest that while superoptimization is inherently difficult, some LLMs can be effective superoptimizers.

Failure Modes. Interestingly, models that are expected to achieve strong results perform (e.g., DeepSeek-R1, GPT-4o) perform poorly on the task of superoptimization, motivating our analysis of their failure modes. We find that DeepSeek-R1 consistently fails to generate valid assembly code, resulting in a 0% compilation rate. DeepSeek-R1 often produces verbose analysis instead of executable code, spending its entire output length on reasoning about instruction semantics and potential optimizations without actually implementing them.

We further analyze the failure modes of GPT-4o, which achieves a high compilation rate (81.0%) but exhibits poor correctness (only 5.0% test pass rate). The primary correctness issues are as follows: (1) missing critical directives and safety setup, such as stack canary initialization and `.cfi_*` metadata, which often lead to runtime crashes; (2) incorrect function call conventions, where repeated system calls like `scanf` are made without proper argument setup, causing undefined behavior; (3)

Model	Compile Pass	Test Pass	Average Speedup
Qwen2.5-Coder-7B (Base)	77.9 ± 0.8%	61.4 ± 0.5%	1.10 ± 0.01×
SuperCoder (GRPO)	95.0 ± 0.0%	94.7 ± 0.6%	1.44 ± 0.07×
SuperCoder (PPO)	96.0 ± 0.0%	95.0 ± 0.0%	1.46 ± 0.12×
SuperCoder (Supervised fine-tuning)	95.5 ± 0.0%	92.5 ± 0.0%	1.39 ± 0.05×

Table 3.3: Performance of the base model and the models trained with RL or supervised fine-tuning. Results include compilation success, test pass rates, and average speedup, reported over 5 runs with 95% confidence intervals.

semantic errors in core computations, including incorrect pointer usage or altered algorithm logic, which produce wrong outputs even when the code runs; and (4) over-simplified stack or register management, resulting in memory errors or invalid control flow. In summary, GPT-4o tends to sacrifice correctness in pursuit of optimization: it generates syntactically valid assembly but frequently violates low-level conventions necessary for correct and reliable execution.

3.5.2 Effectiveness of RL Training

Improvement. Table 3.3 presents the results of RL training, averaged over 5 runs with 95% confidence intervals to provide more statistical confidence in the reported improvements. We select Qwen2.5-Coder-7B-Instruct for RL training due to its strong test pass rate (61.4%) among models. After RL training with PPO, the fine-tuned model SuperCoder attains 95.0% correctness and improves average speedup from 1.10× to 1.46×. Its speedup percentiles are 1.17 ± 0.03× (25th), 1.35 ± 0.04× (50th), and 1.64 ± 0.08× (75th) respectively, outperforming the majority of evaluated models.

PPO versus GRPO. We evaluate both PPO-trained and GRPO-trained models and find their performance to be nearly identical. SuperCoder trained with GRPO attains 94.7 ± 0.6% correctness and 1.44 ± 0.07× average speedup, which is comparable to SuperCoder trained with PPO (95.0 ± 0.0% correctness and 1.46 ± 0.12× average speedup).

3.5.3 Results from Supervised Fine-Tuning and Inference-Time Methods

Best-of-N Sampling. We evaluate best-of-N sampling for three models: claude-opus-4 (the strongest baseline in Table 3.2), Qwen2.5-Coder-7B (base), and SuperCoder (our PPO-trained model). Results are shown in Figure 3.2. Notably, the base model's best-of-8 speedup is close to the PPO-trained model's best-of-1 result, and the RL-trained model itself can still be improved with best-of-N sampling (i.e., from 1.46× in the single-sample setting to 1.93× with best-of-8 sampling).

Supervised Fine-Tuning. We describe our supervised fine-tuning approach in Section 3.3.4. Table 3.2 reports results averaged over five runs with 95% confidence intervals. While supervised fine-tuning improves performance, RL achieves slightly stronger results. We believe that RL is a

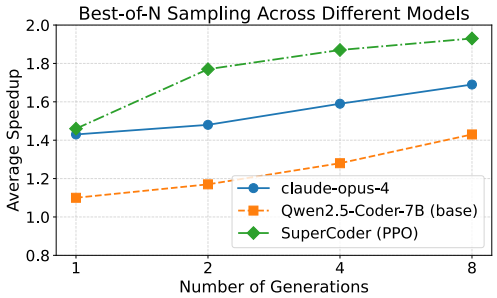


Figure 3.2: Best-of-N sampling results.

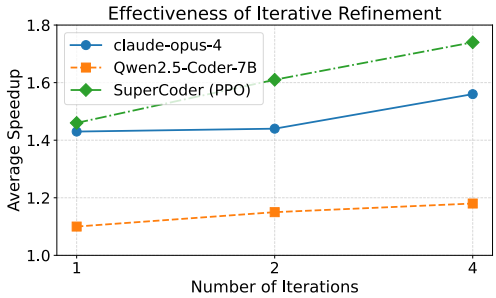


Figure 3.3: Iterative refinement results.

natural fit for the open-ended nature of superoptimization, as RL directly optimizes for correctness and speedup rather than imitating existing examples.

Iterative Refinement. Figure 3.3 shows the results of iterative refinement, where the model receives feedback about compilation failures, test failures, or performance for self-reflection. All three models exhibit improvements as the number of refinement iterations increases, with the effect being most pronounced for our RL fine-tuned model.

3.5.4 Analysis of Program Transformations

To better understand why LLMs can further optimize assembly programs already optimized by industry-standard compilers, we analyze all 200 evaluation programs by comparing the gcc -O3 output with the assembly generated by claude-opus-4 with best-of-8 sampling. We categorize the program transformations into nine types: 1) Loop restructuring covers changes to loop organization, including reordering, unrolling, or altered control flow; 2) Instruction selection captures the use of specialized CPU instructions (e.g., bsrq, popcnt, cmov) in place of longer generic instruction sequences; 3) Algorithmic simplification denotes replacing complex or custom logic with simpler algorithms or standard library routines such as memcmp, strcmp, or atoi; 4) Stack canary removal refers to eliminating stack protection checks and related security instrumentation; 5) Register allocation reflects differences in how registers are assigned, reused, or spilled to memory; 6) Branch elimination involves removing conditional branches by using conditional moves (cmov) or condition-setting instructions (setcc); 7) Address calculation optimization refers to simplifying memory address computations and offset arithmetic; 8) Function call optimization captures substituting heavyweight or checked function variants with simpler equivalents (e.g., __isoc99_scanf vs. scanf); and 9) Arithmetic optimization includes simplifying arithmetic operations such as replacing divisions or multiplications with shifts, or using increment/decrement instructions where applicable.

Figure 3.4 presents a detailed category-wise analysis, reporting the frequency of each transformation category among all transformed programs that pass all tests. For a given optimized version, the model may induce transformations spanning multiple optimization categories. Because performance

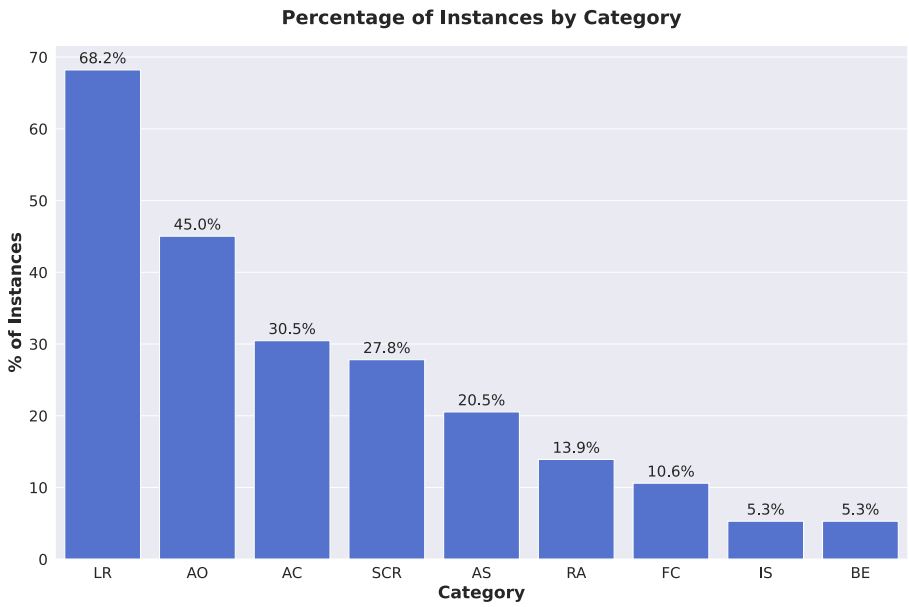


Figure 3.4: Categorization of the program transformations: Loop Restructuring (LR), Arithmetic Optimization (AO), Address Calculation (AC), Stack Canary Removal (SCR), Algorithmic Simplification (AS), Register Allocation (RA), Function Call Optimization (FC), Instruction Selection (IS), Branch Elimination (BE).

optimization typically involves multiple interrelated decisions, it is difficult to automatically isolate which types of program transformations are responsible for observed performance improvements. More broadly, automated analysis and attribution of performance differences at the assembly level remains an open problem.

3.6 Discussion

Prompt Optimization Methods. We experimented with few-shot in-context learning and found that adding more examples does not reliably improve performance and often degrades it (as shown in Table 3.4), consistent with prior observations in code optimization [233]. We also evaluated GEPA [5], an evolutionary prompting framework that uses natural language reflection to derive optimization rules, but observed only minimal gains. We used gpt-4o as the model under evaluation and gpt-5 as the reflection model. GEPA yielded only modest gains: compilation pass increased from 81.0% to 84.0% and test pass from 5.0% to 7.5%, while performance speedup remained essentially unchanged. We suspect this is because assembly optimization requires substantial domain knowledge that is difficult to capture by modifying the prompt alone. Lastly, we also experimented with prompts that include only the C source file, without providing the compiler-generated assembly. We find that omitting the

Model	Shots	Compile Pass (%)	Test Pass (%)	Avg. Speedup
claude-opus-4	0-shot	90.0	51.5	1.43×
claude-opus-4	2-shot	95.0	15.0	1.13×
claude-opus-4	4-shot	95.0	12.5	1.11×
SuperCoder (PPO)	0-shot	96.0	95.0	1.46×
SuperCoder (PPO)	2-shot	94.0	90.5	1.59×
SuperCoder (PPO)	4-shot	93.0	81.0	1.54×
Qwen2.5-Coder-7B (Base)	0-shot	77.9	61.4	1.10×
Qwen2.5-Coder-7B (Base)	2-shot	70.5	35.0	1.10×
Qwen2.5-Coder-7B (Base)	4-shot	80.5	30.5	1.06×

Table 3.4: Comparison of 0-shot, 2-shot, and 4-shot prompting across different models.

C Code

```c
int f(unsigned long x)
{
  int res = 0;
  while (x > 0)
  {
    res += x & 1;
    x >>= 1;
  }
  return res;
}
```

GCC -O3 Output

```
.L0:
 xorl %eax, %eax
 testq %rdi, %rdi
 je .L2
.L1:
 movq %rdi, %rdx
 andl $0x1, %edx
 addq %rdx, %rax
 shrq $0x1, %rdi
 jne .L1
 retq
.L2:
 retq
```

Claude-Opus-4

```
.L0:
 popcnt %rdi, %rax
 retq
```

Figure 3.5: Case study comparing the C code, baseline assembly produced by gcc -O3, and optimized assembly generated by Claude-Opus-4. The model successfully replaces the loop with the specialized hardware instruction `popcnt`, resulting in a significantly more concise implementation.

assembly as a starting point leads to substantially worse results. For example, SuperCoder, which attains 95.0% correctness with the assembly baseline, fails to produce any compilable code without it. Similar degradation occurs for other models, such as llm-compiler-13b and Claude models. These results indicate that, at least in their current state, LLMs can act as superoptimizers building on compiler outputs, but cannot replace compilers themselves.

Alternative Reward Function Design. Besides the reward function presented in Section 3.3.3, we also evaluate an alternative design. The original design assigns zero reward whenever any test fails. In contrast, the alternative assigns (i) a reward of -1 if the program fails to compile, (ii) a partial reward equal to the fraction of passed tests if only some tests succeed, and (iii) a reward of $1 + \alpha \cdot \text{speedup}$ once all tests pass. Training the base model with this design yields an average speedup of 1.38×. Varying the scaling factor (5 or 10) has a negligible effect, and the result remains

GCC -O3 Output

```
    ...
    testl   %eax, %eax
    jle     .L2
    movl    %eax, %r8d
    jmp     .L5
    .p2align 4,,10
    .p2align 3
.L4:
    subl    $1, %r8d
    je      .L2
.L5:
    movl    %ecx, %eax
    cltd
    idivl   %r8d
    testl   %edx, %edx
    jne     .L4
    movl    %r8d, %edx
    leaq    .LC1(%rip), %rsi
    movl    $1, %edi
    xorl    %eax, %eax
    call    __printf_chk@PLT
.L2:
    movq  8(%rsp), %rax
    subq  %fs:40, %rax
    jne .L14
    xorl  %eax, %eax
    addq  $24, %rsp
    .cfi_remember_state
    .cfi_def_cfa_offset 8
    ret
    ...

```

Claude-Opus-4

```
    ...
    testl   %eax, %eax
    jle     .L2
.L3:
    movl    %ecx, %edx
    movl    %eax, %esi
    movl    %edx, %eax
    cltd
    idivl   %esi
    testl   %edx, %edx
    je      .L4
    decl    %esi
    movl    %esi, %eax
    jg      .L3
.L2:
    xorl    %eax, %eax
    addq    $24, %rsp
    ret
.L4:
    leaq  .LC1(%rip), %rdi
    xorl  %eax, %eax
    call  printf@PLT
    xorl  %eax, %eax
    addq  $24, %rsp
    ret
    .size main, .-main

```

Figure 3.6: Case study comparing the baseline assembly code snippet produced by gcc -O3 and the optimized assembly code snippet generated by Claude-Opus-4. Claude-Opus-4 eliminates the entry-path unconditional jump and alignment padding by fusing GCC's two-block loop into a single, simpler control-flow structure, calls `printf@PLT` directly (a simpler function variant without security checks), and removes gcc's stack-protector canary check.

worse than the 1.46× achieved by SuperCoder with the original reward. This suggests that directly optimizing for the terminal speedup reward is more effective.

Case Studies. We illustrate in Figure 3.5 and Figure 3.6 two representative examples where an LLM discovers optimizations beyond the reach of a state-of-the-art compiler. The first example is about instruction selection. In the first example, the original C function computes the population count (i.e., the number of set bits) by repeatedly shifting the input and accumulating its least significant bit. The assembly produced by gcc -O3 preserves this loop structure. In contrast, Claude-opus-4 produces an efficient implementation that replaces the entire loop with a single `popcnt` instruction, performing the same computation in one operation. The second example involves loop restructuring,

along with function call optimization by replacing checked function variants with simpler equivalents.

Limitation and Potential Directions. Our evaluation relies on test-based validation, which is common in prior work [233, 64]. While effective in practice, this approach would be more faithful with access to a formal equivalence checker for assembly programs. Existing state-of-the-art equivalence checkers for assembly [216] cannot handle programs with the rich control flow present in our dataset. To mitigate this limitation, our evaluation achieves 96.2% line coverage, substantially reducing the risk of undetected errors. Encouragingly, recent work has begun developing formal verification tools for GPU kernels [66], partly motivated by advances in LLM-generated GPU code. We hope that our work further motivates the development of more general and scalable verification tools for assembly programs. In addition, most superoptimization research has centered on x86-64. Extending the methodology to other targets such as ARM, RISC-V, or GPU kernels could greatly broaden its applicability.

3.7 Conclusion

We investigated whether LLMs can act as superoptimizers, generating assembly programs that outperform code already optimized by industry-standard compilers. To support this study, we introduced the first large-scale benchmark of 8,072 assembly programs. Evaluating 23 models revealed the difficulty of the task, with most failing to achieve meaningful gains. By fine-tuning with reinforcement learning, our model SuperCoder improved both correctness and performance, reaching 95.0% test pass rate and an average speedup of 1.46× over gcc -O3. We also show that Best-of-N sampling and iterative refinement can bring additional improvement. These results demonstrate, for the first time, that LLMs can be applied as superoptimizers for assembly code, opening new opportunities for performance optimization beyond compiler heuristics.

Chapter 4

Accelerating Verification via Invariant Synthesis

4.1 Introduction

Program verification aims to provide formal guarantees that software behaves as intended, with applications in many safety-critical domains [74, 176]. A long-standing challenge in this area, studied for more than four decades, is the automatic discovery of loop invariants. In this chapter, we investigate whether large language models (LLMs) can accelerate program verification by generating useful loop invariants.

Loop invariants are conditions that hold before and after each loop iteration, and they are central to deductive program verification. To accelerate program verification, loop invariants must not only be correct but also sufficiently strong to prove the assertions. Generating correct invariants is relatively easy, since any universally true condition qualifies. However, only strong invariants can reduce verification effort and lead to a speedup. For example, in Figure 4.1, the invariant $x > 0$ is correct but not strong enough to prove the final assertion $x \neq 700$, whereas $x \equiv 3 \pmod 7$ is both correct and sufficiently strong.

Discovering such invariants is difficult and undecidable in general, which has motivated a long line of research. Traditional approaches include constraint solving [46, 100], dynamic analysis [147], etc. Since invariant discovery is challenging, researchers have tried a variety of learning-based methods [151, 73]. Building on this progression, the strong capabilities of LLMs in code generation and program reasoning [13, 34, 267] naturally motivate a systematic evaluation of their potential for invariant discovery.

The first work [204] to evaluate LLMs for invariant generation has two key limitations. One limitation is that it only checks correctness without assessing whether invariants are strong enough

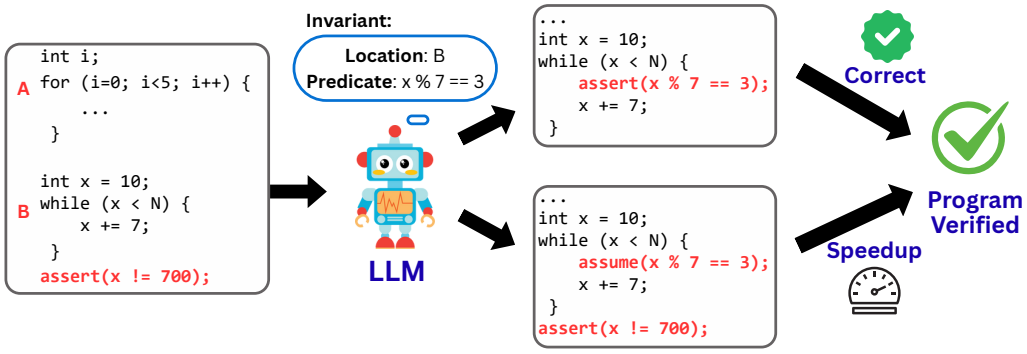


Figure 4.1: Illustration of Quokka's evaluation pipeline. The LLM proposes an invariant by specifying a program location and predicate (e.g., location B with `x % 7 == 3`). The verification procedure then incorporates this invariant to prove the property `x != 700` using two verifier queries, and we measure the resulting speedup relative to a baseline without LLM assistance.

to accelerate verification. Another limitation is that correctness is determined by directly comparing against Daikon [70], a dynamic analysis tool that infers invariants from test executions rather than formal verification. This approach is unsound: Daikon's invariants may themselves be incorrect. Moreover, semantically equivalent invariants (e.g., $a > 0$ vs. $a \geq 1$ for integers) can be incorrectly rejected as wrong. Consequently, prior work cannot reliably evaluate either the correctness or the practical utility of LLM-generated invariants.

A series of follow-up works have proposed LLM-based verifiers. Instead of evaluating LLMs in isolation, these efforts develop verification frameworks powered by LLMs. This line of work predominantly adopts sequential, iterative, and relatively complex (compared to our own approach) processes with customized algorithms to handle LLM-generated invariants. For instance, LaM4Inv [277] uses a "query-filter-reassemble" strategy that filters and combines model-generated predicates through complex logical operations (conjunctions or disjunctions) to construct valid loop invariants, rather than using them directly. LOOPY [132] employs a customized Houdini algorithm with iterative refinement to filter candidate invariants. LEMUR [278] develops a backtracking and sequential algorithm. While these approaches demonstrate the potential of LLM-based verification, their complexity raises a question: can we design a simpler, principled, and more effective algorithm for LLM-based verifiers?

In this chapter, we introduce Quokka, a framework for LLM-based invariant synthesis that accelerates program verification. Quokka provides a sound approach to evaluate LLMs' capabilities in generating invariants while achieving state-of-the-art speedup results compared with prior LLM-based verifiers. We formalize our methodology as a proof calculus, providing a rigorous foundation for building LLM-based verifiers. Instead of checking the correctness of invariants based on string-matching, we adopt a verifier-based approach that directly queries the underlying solver. Unlike previous work that designs complex, highly customized algorithms, Quokka employs a simple and

principled verification procedure. Our approach enables both soundness guarantees and precise measurement of performance speedup, allowing us to systematically evaluate whether LLM-generated invariants can accelerate verification in practice.

To support systematic comparison across solvers and LLMs, we construct a dataset of 866 instances derived from the most recent edition of the software verification competition SV-COMP [22]. To the best of our knowledge, this is the largest evaluation dataset for LLM-based verifiers to date, compared with datasets used in previous work. Quokka consistently outperforms all prior LLM-based verifiers. When comparing against LEMUR (the previous best LLM-based verifier with the same underlying solver), Quokka achieves speedups of at least 1.2× (i.e., verification completes at least 20% faster than the baseline solver) on 81 instances compared to LEMUR's 39 instances. For more substantial speedups of at least 2.0× (i.e., verification completes at least twice as fast), Quokka reaches 51 instances compared to LEMUR's 22 instances.

We evaluate 9 state-of-the-art LLMs, including GPT models from OpenAI, Claude models from Anthropic, and models from the Qwen and LLaMA families. Our experiments reveal that gpt-5.2 achieves the strongest speedup results among all evaluated models. Beyond model evaluation, we investigate techniques to enhance invariant generation capabilities. We construct a training dataset of 3589 instances using a verifier-based filtering approach that ensures data quality. Our experiments demonstrate that both supervised fine-tuning and Best-of-N sampling yield measurable improvements in accelerating verification. Specifically, Best-of-8 sampling achieves speedups of at least 1.2× on 99 instances, representing a 22% improvement over single-sample generation, which achieves such speedups on only 81 instances.

In summary, our contributions are as follows:

- We introduce an algorithm for LLM-based invariant synthesis that provides sound evaluation and achieves the best speedup results compared with prior LLM-based verifiers.
- We construct a benchmark of 866 instances, and conduct an evaluation of 9 state-of-the-art LLMs across multiple model families.
- We construct training data and demonstrate that both supervised fine-tuning and Best-of-N sampling yield measurable improvements in accelerating verification.

4.2 Related Work

Traditional Methods for Program Invariant Generation. A long line of research has explored invariant synthesis using traditional techniques without machine learning, including model checking [85, 142, 113, 258], abstract interpretation [135, 47, 49, 48], constraint solving [96, 100], Craig interpolation [126, 183], and syntax-guided synthesis [77]. Prior work evaluating LLM-generated invariants [204] has relied on Daikon [70], a tool for dynamic invariant detection [67, 147]. Daikon

executes the program, observes runtime values, and reports properties that consistently hold over the observed executions. However, such invariants may fail to generalize to all possible executions, thereby compromising soundness. Our approach instead employs a verifier-based algorithm relying on UAutomizer [219] that ensures soundness.

Learning-Based Method for Invariant Generation. Machine learning based techniques have been widely adopted in invariant synthesis, including decision tree [91, 92, 73, 212, 290], support vector machine [151, 227], reinforcement learning [234, 307], and others [226, 215, 304]. More recently, large language models have demonstrated strong capabilities in reasoning about code and logic [264, 267, 271], giving rise to a series of work that explore using LLMs for finding invariants. [204] is the first pioneering work that evaluates LLMs' capabilities in finding invariants, but it is not a sound evaluation. Various techniques have been proposed to couple LLMs with symbolic solvers, including ranking LLM-generated invariants [31], the "query-filter-reassemble" strategy of LaM4Inv [277], the back-tracking algorithm in LEMUR [278], and Loopy's integration of the classic Houdini algorithm [132]. On the dataset side, [161] introduces a rule-based method for constructing a fine-tuning corpus, which differs from our verifier-based approach. In contrast, our work provides a simple and sound evaluation procedure for assessing LLM-generated invariants and investigates how both fine-tuning and Best-of-N sampling can enhance LLM performance in invariant synthesis.

4.3 Method

4.3.1 Preliminary

We formalize the task of loop invariant synthesis using standard Hoare logic [112]. A Hoare triple $\{P\}\, S\, \{Q\}$ specifies that if the precondition P holds before executing a statement S, then the postcondition Q will hold after its execution. In the context of loops, an invariant I is a logical proposition that summarizes the state of the program at each iteration, and it is the key to proving the validity of Hoare triples involving loops. For a loop of the form `while` B `do` S, the goal of invariant synthesis is to identify a loop invariant I that satisfies the following inference rule:

$$\frac{P \Rightarrow I \quad \{I \wedge B\}\, S\, \{I\} \quad I \wedge \neg B \Rightarrow Q}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{Q\}}$$

Here, P is the precondition, Q is the postcondition, B is the loop condition, and S is the loop body. Intuitively, the inference rule requires that the invariant I holds at the beginning of the loop ($P \Rightarrow I$), is preserved across every iteration of the loop body ($\{I \wedge B\}\, S\, \{I\}$), and upon termination ensures the postcondition ($I \wedge \neg B \Rightarrow Q$).

Invariant synthesis amounts to generating a logical summary I that is both *correct*, meaning it can be verified, and *strong*, meaning it enables verification of the final assertion. Weak but correct

invariants contribute little, leaving most of the reasoning to the verifier, whereas strong invariants narrow the search space of program states, reduce solver effort, and yield substantial speedups.

4.3.2 Verifier-Based Algorithm

We formalize our verifier-based procedure for assessing candidate invariants. Let P denote a program. A *property* is written as $p = \langle \varphi, \ell \rangle$, where φ is a state predicate and ℓ is a program location. For a finite set A of properties, let $\mathrm{Asm}(P, A)$ be the program obtained from P by inserting `assume` statements for all elements of A. An execution of $\mathrm{Asm}(P, A)$ that reaches a location where an assumption is violated terminates immediately. We write $P \models_A p$ to indicate that all executions of $\mathrm{Asm}(P, A)$ satisfy the assertion p. The notation $P \models p$ abbreviates $P \models_\varnothing p$. Since assumptions restrict behaviors, if $P \not\models_A p$ for some A, then necessarily $P \not\models p$.

We assume access to a verifier

$$V(P, A, p) \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\},$$

which returns either **T** (proved), **F** (refuted), or **U** (inconclusive). The verifier is required to be sound on conclusive outcomes:

$$V(P, A, p) = \mathbf{T} \;\Rightarrow\; P \models_A p,$$

$$V(P, A, p) = \mathbf{F} \;\Rightarrow\; P \not\models_A p.$$

No completeness is assumed for **U**, which may arise from timeouts or incompleteness of the underlying verifier.

The verification task specifies a target property $p^\star = \langle \varphi^\star, \ell^\star \rangle$. Given P and $p^\star$, a large language model proposes a *candidate invariant* $q = \langle \psi, \ell \rangle$, typically at a loop header. To evaluate the utility of q, the procedure issues two verifier queries:

$$d_a := V(P, \varnothing, q)$$

$$d_b := V(P, \{q\}, p^\star)$$

The query d_a verifies whether q is a correct invariant, while the query d_b determines whether the target property holds under the assumption that q is true. An illustration of these two verifier queries is provided in Figure 4.1: d_a corresponds to `assert(x % 7 == 3)` (upper subfigure), and d_b corresponds to verifying the final assertion with the assume statement inserted (lower subfigure). The outcome of the procedure is expressed as a judgment

$$P \;\Rightarrow\; \langle p^\star, q \rangle \;\Downarrow\; d \qquad \text{with} \quad d \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}.$$

The interpretation is as follows: if the judgment yields $\mathbf{T}$, then $p^\star$ is established on P; if it yields $\mathbf{F}$, then $p^\star$ is refuted; and if it yields $\mathbf{U}$, the attempt is inconclusive.

The inference rules defining this judgment are given below. Each rule specifies one possible derivation of the outcome, depending only on the responses of the verifier.

$$\frac{V(P, \{q\}, p^\star) = \mathbf{F}}{P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; \mathbf{F}} \text{ (DEC-FALSE)}$$

$$\frac{V(P, \varnothing, q) = \mathbf{T} \qquad V(P, \{q\}, p^\star) = d \qquad d \neq \mathbf{F}}{P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; d} \text{ (DEC-PROP)}$$

$$\frac{V(P, \varnothing, q) \neq \mathbf{T} \qquad V(P, \{q\}, p^\star) \neq \mathbf{F}}{P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; \mathbf{U}} \text{ (DEC-}\mathbf{U}\text{)}$$

Rule DEC-FALSE captures short-circuit refutation: if the goal fails even in the restricted program $\mathrm{Asm}(P, \{q\})$, then it is false on the original program P. Rule DEC-PROP implements the prove-then-use strategy: once the candidate invariant q is established, the outcome is exactly the verifier's answer on the goal under the assumption q, restricted to $d \in \{\mathbf{T}, \mathbf{U}\}$ so as not to overlap with DEC-FALSE. Rule DEC-$\mathbf{U}$ gives explicit conditions for inconclusiveness: the goal is not refuted under q and q is not established as an invariant.

Theorem (Decision Soundness). If $P \Rightarrow \langle p^\star, q\rangle \Downarrow \mathbf{T}$ is derivable, then $P \models p^\star$. If $P \Rightarrow \langle p^\star, q\rangle \Downarrow \mathbf{F}$ is derivable, then $P \not\models p^\star$.

Proof. For the $\mathbf{T}$ case, the final rule must be DEC-PROP with $d = \mathbf{T}$. The premises yield $V(P, \varnothing, q) = \mathbf{T}$ and $V(P, \{q\}, p^\star) = \mathbf{T}$. By verifier soundness, $P \models q$ and $P \models_{\{q\}} p^\star$. Since q holds on all executions of P, introducing the assumption $\{q\}$ does not remove executions relevant to $p^\star$; thus $P \models p^\star$. For the $\mathbf{F}$ case, the final rule must be DEC-FALSE. Its premise $V(P, \{q\}, p^\star) = \mathbf{F}$ implies $P \not\models_{\{q\}} p^\star$ by soundness. Assumptions restrict behaviors; hence, a violation under assumptions entails a violation without them, yielding $P \not\models p^\star$. □

This theorem establishes that whenever the calculus derives a conclusive outcome, that outcome is correct. The inconclusive case $\mathbf{U}$ is deliberately conservative: it makes no claim about the truth or falsity of the property and safely captures verifier incompleteness or timeouts.

4.3.3 Implementation

We describe the implementation of our verifier-based evaluation framework. Given a program P and target property $p^\star$, the system evaluates the generated invariants according to the algorithm. Each

invariant $q = \langle \psi, \ell \rangle$ consists of a program location ℓ and predicate ψ.

Syntactic Validation. Before invoking the verifier, we apply syntactic checks to the generated predicate ψ. These checks ensure that ψ can be safely interpreted as a state predicate and that its insertion as an assumption does not alter the program state. For instance, expressions that update variables (e.g., `a += 1`) are rejected. Only Boolean conditions over the program state are accepted.

Parallel Verifier Queries. For each candidate q, the procedure issues two verifier queries, namely $d_a = V(P, \varnothing, q)$ to check whether q is an invariant and $d_b = V(P, \{q\}, p^\star)$ to check whether the target holds under the assumption q. These queries are executed in parallel in our implementation, which reduces latency and enables speedup when the proposed invariant is useful for verification. The final outcome is then derived exactly according to the decision calculus.

4.3.4 Supervised Fine-Tuning and Best-of-N Sampling

Below, we discuss how we construct our dataset for fine-tuning and the way we perform Best-of-N sampling.

Synthetic Dataset Generation. To construct the synthetic dataset, we prompt GPT-4o. The prompt template takes three seed programs as examples and instructs the model to synthesize a new C program that is compilable and contains both loops and assertions. To obtain a diverse and large pool of candidates, we repeatedly invoke the model with different seed programs. To avoid data leakage, these seed programs are randomly drawn from the SV-COMP pool [22] that is *disjoint* from our evaluation set. Although the prompt requests compilable programs with loops and assertions, the LLM-generated programs may fail to compile, include assertions that do not hold, and contain multiple assertions. For programs with multiple assertions, we split them into separate instances, each retaining only a single assertion while preserving all loop structures (ensuring at least one loop per instance). We then run UAutomizer on every program and discard any instance that is non-compilable or whose assertion is invalid. This filtering step ensures the quality of the dataset, resulting in 3589 synthetic programs.

Extract Invariants Generated from UAutomizer. When running UAutomizer to prove the assertions in the synthetic programs, the tool also emits the invariants it discovers. From its output, we extract the loop invariants. Each extracted invariant includes its program location and its predicate. Although each program contains exactly one assertion, it may include multiple loops, so a single program can yield multiple loop invariants, all associated with the same assertion. We pair each program with each of its corresponding loop invariants to form our training dataset. With the ground-truth invariants, we perform supervised fine-tuning using LoRA [115].

Dataset	Avg. #Lines	#Instances
Evaluation	53	866
Training	42	3589

Table 4.1: Dataset statistics.

Best-of-N Sampling. Best-of-N sampling is an inference-time strategy in which multiple candidate programs are generated, and the most effective one is selected, a technique shown to improve performance on code-generation tasks [68]. In our setting, the best candidate is the invariant that yields the largest speedup, i.e., the one whose algorithm finishes earliest. As described in Section 4.3.2, evaluating a single candidate requires two verifier queries issued in parallel; therefore, Best-of-N sampling evaluates 2N verifier queries concurrently.

4.4 Experimental Setup

Dataset. We construct our evaluation benchmark from SV-COMP [22], a standard competition in software verification, focusing on problems that require loop invariant synthesis. We collect a pool of 866 instances and run the state-of-the-art non-LLM verifier UAutomizer [219] with a 600-second timeout to record per-instance solving times. The process is fully automated, with no manual filtering or cherry-picking. We use this timeout to enable meaningful measurement of verification speedups. In contrast to prior work [278], which evaluates only instances that the baseline cannot solve within this time budget, our approach requires knowing the baseline runtime to quantify relative improvements. Restricting to instances whose baseline runtime is observable allows us to report concrete speedup factors, rather than only counts of additionally solved problems. To the best of our knowledge, this dataset is the largest used in prior evaluations of LLM-based verifiers. Also, the programs in our dataset are significantly longer, with an average of 53 lines of code, compared to only 22 for LaM4Inv [277], 23 for LEMUR [278], and 27 for Loopy [133]. Manual inspection shows that our dataset contains features such as multiple loops, functions, arrays, and pointers, which are largely absent from prior datasets. This makes Quokka a more challenging and realistic benchmark that better distinguishes solver performance.

For training data, we construct our training data with synthetic data generation filtered by the solvers, as described in Section 4.3.4. We make sure that the seed programs used for synthetic data generation are disjoint from our evaluation dataset, mitigating the leakage issue. The statistics of our dataset are shown in Table 4.1.

Metrics. We evaluate LLMs along two dimensions: (1) the correctness of the generated invariants and (2) the performance improvements they provide. Correctness is determined by the solver using a timeout set to 1.2× the original solving time for each problem. This slack allows the solver additional

Model	# Correct Invariants	# $\textbf{Fast}_{1.2}$	# $\textbf{Fast}_{1.4}$	# $\textbf{Fast}_{1.6}$	# $\textbf{Fast}_{1.8}$	# $\textbf{Fast}_{2.0}$
Qwen2.5-7B	419	34	26	23	20	18
o3	550	38	33	30	24	23
Llama-3.1-8B	342	43	35	32	29	27
claude-opus-4.1	487	56	45	38	35	31
Qwen2.5-72B	500	63	49	45	37	33
claude-opus-4.5	689	69	53	46	39	34
gpt-5.1	694	56	46	41	40	37
claude-sonnet-4	620	68	58	51	45	41
gpt-5.2	710	81	65	60	56	51

Table 4.2: Performance of different LLMs when using Quokka. We report the number of instances with verified-correct invariants and the number of instances achieving speedups of at least 1.2×, 1.4×, 1.6×, 1.8×, and 2.0× over UAutomizer.

time to assess the correctness of model-generated invariants, reducing inconclusive outcomes caused by timeouts. All speedup-related measurements are reported relative to the original solver, which serves as the baseline. We report speedup using the metric of $\#\texttt{Fast}_p$, defined as the number of instances for which verification completes at least $p\times$ faster than the baseline, with $p \in \{1.2, 1.4, \ldots\}$. Additionally, we also report the number of instances solved under varying timeout budgets.

Choice of Solvers. We use the state-of-the-art non-LLM-based solver UAutomizer [219] as our base solver. Our methodology requires that the solver can verify programs without externally-provided invariants, enabling us to measure speedup relative to a functional baseline. While prior work has explored other solvers such as Frama-C [53] and ESBMC [88], we select UAutomizer for two key reasons. First, UAutomizer represents the current state-of-the-art in automated verification, outperforming ESBMC on standard benchmarks in SV-COMP. Second, unlike Frama-C, which requires externally-provided invariants and cannot proceed without them, UAutomizer has built-in invariant synthesis capabilities that allow it to verify programs autonomously. To demonstrate the generality of Quokka's methodology across different solvers, we also run experiments with ESBMC, and we find that Quokka can accelerate ESBMC as well.

Models. We benchmark Claude models from Anthropic, GPT models from OpenAI, and models from the Qwen and LLaMA families [121, 296, 252].

Hardware and OS. Experiments were conducted on a node with Intel E5-2640 v4 CPUs, 256G main memory, running Ubuntu 20.04 LTS.

4.5 Results

4.5.1 Results of LLMs

Table 4.2 presents the performance of different LLMs when using Quokka. The results are sorted by the number of instances achieving at least 2.0× speedup (# $\text{Fast}_{2.0}$), which ranges from 18 instances for Qwen2.5-7B to 51 instances for gpt-5.2. Overall, the table demonstrates substantial variation in both correctness and speedup performance across different models, with larger and more recent models generally achieving better results.

Effectiveness of Quokka. Our approach demonstrates the speedup capabilities across all evaluated models. Even smaller models like Qwen2.5-7B achieve speedups on 34 instances at the 1.2× threshold, while the best-performing model gpt-5.2 accelerates 81 instances at this level and achieves 2.0× or greater speedup on 51 instances. These results demonstrate that Quokka can effectively leverage LLMs to generate invariants that meaningfully accelerate verification. The consistent speedup gains across different model families, from 7B parameter models to frontier models, and the progressive improvement with model scale underscore the effectiveness of our approach.

Correctness vs. Performance Gap. A striking observation is the substantial gap between generating correct invariants and achieving meaningful speedups. For example, gpt-5.2 produces 710 correct invariants but achieves 2.0× speedup on only 51 instances, while claude-opus-4.5 generates 689 correct invariants yet delivers 2.0× speedup on merely 34 instances. This gap highlights that correctness alone is insufficient: invariants must also be strong enough to meaningfully accelerate verification, requiring not only logical correctness but also strategic insight into which invariants will most effectively guide the solver.

Model Inference Time. We include model inference time in our evaluation to provide a fair and realistic assessment of end-to-end performance. We find that smaller models such as Llama-3.1-8B and Qwen2.5-7B exhibit negligible generation times (0.3–0.4 seconds), while most models complete inference within 3 seconds. The o3 model has the highest overhead at 8.1 seconds on average. Importantly, unlike LOOPY [133], which invokes the LLM up to 15 times per program and thus spends a substantial portion of its time budget on API calls, our approach requires only a single LLM invocation per program. Consequently, LLM inference accounts for only a small fraction of total runtime. Furthermore, our evaluation dataset contains sufficiently challenging programs (with solving times up to 600 seconds) such that solver time naturally dominates inference overhead, making the comparison meaningful even when inference latency is included.

Prompting-Based Methods. We investigate the impact of different prompting strategies on invariant generation quality using gpt-5.2. We evaluate 0-shot, 2-shot, 4-shot, and chain-of-thought

Method	# Fast$_{1.2}$	# Fast$_{1.4}$	# Fast$_{1.6}$	# Fast$_{1.8}$	# Fast$_{2.0}$
LaM4Inv	1	1	1	1	1
LOOPY	8	7	5	3	1
LEMUR-original	19	16	13	12	10
LEMUR-gpt-5.2	39	30	28	25	22
Quokka-gpt-5.2	81	65	60	56	51

Table 4.3: Comparison of speedup performance across different methods. We report the number of instances achieving speedups of at least 1.2×, 1.4×, 1.6×, 1.8×, and 2.0× over UAutomizer.

(CoT) prompting. Few-shot prompting does not consistently improve performance, while CoT prompting yields the weakest results, likely due to increased generation time without proportional quality gains. These findings indicate that simple zero-shot prompting is most effective for invariant synthesis.

4.5.2 Comparison with Prior LLM-based Verifiers

Table 4.3 compares the speedup performance of Quokka against prior LLM-based verifiers across different speedup thresholds, while Figure 4.2 shows the number of solved instances over time for each method.

Quokka Outperforms Prior LLM-based Verifiers. Quokka with gpt-5.2 achieves state-of-the-art results across all speedup metrics. Among prior work, LEMUR [278] represents the strongest baseline. To ensure a fair comparison, we enhance LEMUR by replacing its original model (gpt-4) with gpt-5.2, creating LEMUR-gpt-5.2. Even with this enhancement, Quokka-gpt-5.2 significantly outperforms LEMUR-gpt-5.2: at the 1.2× speedup threshold, Quokka-gpt-5.2 accelerates 81 instances versus 39 for LEMUR-gpt-5.2 and 19 for LEMUR-original. At 2.0× speedup, Quokka-gpt-5.2 achieves 51 instances compared to 22 for LEMUR-gpt-5.2 and 10 for LEMUR-original. LaM4Inv [277] and LOOPY [133] demonstrate minimal speedup capabilities, achieving at most 1 and 8 instances respectively at the 1.2× level. These results demonstrate that Quokka establishes a new state-of-the-art for LLM-based verification acceleration.

Differences in Base Solvers. Different LLM-based verification frameworks are built on different base solvers: Quokka and LEMUR use UAutomizer, LOOPY uses Frama-C [53], and LaM4Inv uses ESBMC [88]. The choice of base solver significantly impacts the performance, as stronger solvers provide a better foundation for speedup. Since UAutomizer represents the current state-of-the-art solver, we believe future work should prioritize developing LLM-based verifiers atop state-of-the-art solvers such as UAutomizer.

Algorithmic Simplicity and Parallelism. Prior work predominantly adopts sequential, iterative, and relatively complex (compared to our own approach) processes with customized algorithms to handle LLM-generated invariants. In contrast, Quokka employs a simple, verifier-based, and parallel approach that directly leverages the base solver's capabilities.

LaM4Inv uses a "query-filter-reassemble" strategy that filters and combines model-generated predicates through a complex algorithm, rather than using them directly. LOOPY employs a highly-customized Houdini algorithm with iterative fixing to filter candidates. LEMUR is closest to our approach, also directly using the base solver to verify the correctness of invariants, but follows a purely sequential decision process with backtracking. In contrast, Quokka directly uses LLM-generated invariants without decomposition or reassembly, and exploits parallelism by issuing the verification queries in parallel. This design reduces implementation complexity while enabling more efficient hardware utilization, contributing to the superior speedup performance demonstrated in our experiments.

Detailed Comparison with LEMUR. To better understand the performance differences between LEMUR and Quokka, we analyze their implementation strategies and examine the instances where Quokka outperforms LEMUR. LEMUR allows LLMs to insert invariants at multiple locations (both before and at the beginning of loops) and generates multiple candidate invariants for each location. It then sequentially evaluates each candidate, verifying both the original assertion and the invariant itself. If verification times out within 30 seconds, LEMUR recursively invokes the LLM again to generate additional invariants. In contrast, Quokka adopts a simpler strategy: it restricts invariant placement to the beginning of loops, asks the LLM to select a single location (when multiple loops exist), and generates only one invariant. The two verification queries (for the assertion and the invariant) are then executed in parallel without recursion or additional LLM invocations. Among the 353 instances that Quokka solves but LEMUR does not, 129 instances (36.5%) contain multiple loops. We hypothesize that for programs with multiple loops, LEMUR's exhaustive search strategy becomes less effective due to the combinatorially larger space of invariant placements and candidates it must explore. In contrast, Quokka's focused approach (i.e., requiring the LLM to commit to a single location and invariant) appears more effective in these complex scenarios. We identify three key advantages of our design. First, parallel execution of verification queries reduces wall-clock time by simultaneously checking both the assertion and the invariant. Second, constraining the LLM to select a single location and invariant forces more deliberate reasoning about which invariant will be most impactful, rather than relying on exhaustive search over many candidates. Third, eliminating recursion and repair loops reduces both implementation complexity and the risk of execution failures (which we observe in LEMUR's logs). Together, these design choices enable Quokka to achieve superior performance while maintaining a simpler and more robust implementation.

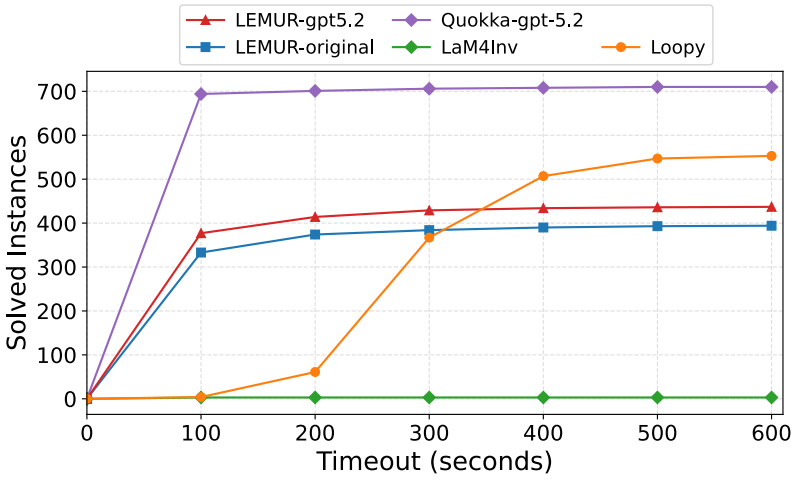


Figure 4.2: Number of instances solved by different methods over varying timeout budgets.

Model	# Incorrect Invariants	# Goal Timeout	# Invariant Timeout	# Both Timeout
gpt-5.1	78	96	245	110
claude-sonnet-4	110	160	209	145
gpt-5.2	56	139	230	121

Table 4.4: Failure mode breakdown for the top three models on Quokka. *Incorrect Invariants*: the invariant is refuted by the solver. *Goal Timeout*: the invariant is verified, but proving the assertion under it times out. *Invariant Timeout*: verifying the invariant itself times out. *Both Timeout*: both verification queries time out.

Model	# Correct	# $\mathbf{Fast}_{1.0}$	# $\mathbf{Fast}_{1.2}$	# $\mathbf{Fast}_{1.4}$	# $\mathbf{Fast}_{1.6}$	# $\mathbf{Fast}_{1.8}$	# $\mathbf{Fast}_{2.0}$
Qwen2.5-7B (base)	419	116	34	26	23	20	18
Qwen2.5-7B (finetuned)	431	141	39	27	26	24	20

Table 4.5: Performance comparison of base and fine-tuned Qwen2.5-7B. Fine-tuning leads to modest improvements in both correctness and speedup metrics.

4.5.3 Failure Mode Analysis

To understand why many invariants fail to accelerate verification in Quokka, we analyze failure modes for the top three models. We categorize failures into four types: 1) *Incorrect Invariants*: the invariant is refuted by the solver; 2) *Goal Timeout*: the invariant is verified, but proving the assertion under it times out; 3) *Invariant Timeout*: verifying the invariant itself times out; 4) *Both Timeout*: both verification queries time out. The results in Table 4.4 reveal that while some failures stem from incorrect invariants, the dominant issue is timeout-related. This suggests that current models lack an understanding of what makes verification queries tractable for symbolic solvers. Future work should explore training reward models or other mechanisms to help models internalize solver

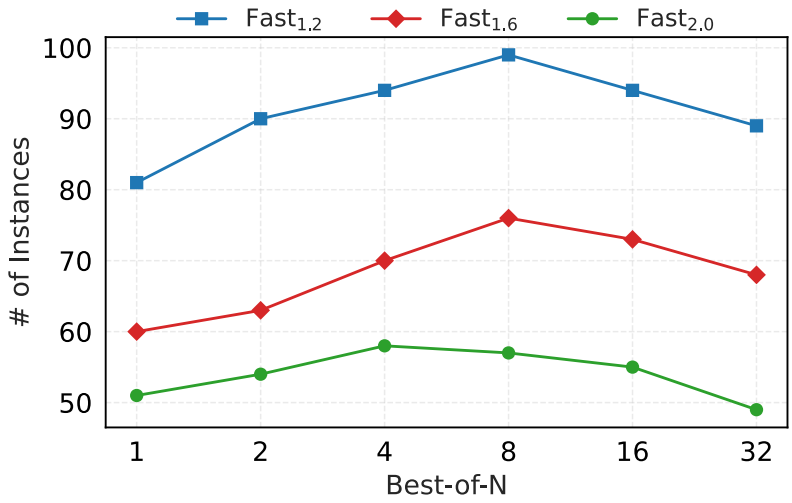


Figure 4.3: Effect of Best-of-N sampling on the number of instances under different Fast$_p$ settings.

difficulty, enabling them to propose invariants that genuinely simplify verification by decomposing hard problems into easier subproblems rather than inadvertently generating verification conditions that are as difficult or harder to solve than the original proof goal.

4.5.4 Results of Fine-Tuning and Best-of-N Sampling

Modest Improvement from Supervised Fine-Tuning. We perform supervised fine-tuning using LoRA [115] on Qwen2.5-7B for 3 epochs. As shown in Table 4.5, fine-tuning yields modest improvements: correct invariants increase from 419 to 431, and instances achieving meaningful speedups improve across all thresholds. For example, Fast$_{1.0}$ improves from 116 to 141, and Fast$_{1.2}$ improves from 34 to 39. These results suggest that domain-specific fine-tuning can improve both correctness and speedup, though gains remain moderate.

Benfits of Best-of-N Sampling. We evaluate Best-of-N sampling by generating multiple invariant candidates from gpt-5.2 at temperature 0.7 and selecting the one yielding the fastest verification time. As shown in Figure 4.3, increasing N from 1 to 4 consistently improves speedup across all Fast$_p$ thresholds. However, performance plateaus beyond N=8 due to hardware constraints: our 20-core CPU must verify 2N queries in parallel, and when N exceeds 10, resource contention may negate sampling benefits. Memory limits can further constrain scalability. While Best-of-N sampling is effective, its benefits are hardware-bounded, though modest values of N yield meaningful improvements.

4.6 Conclusion

This chapter introduced Quokka, a simple yet effective framework for LLM-based invariant synthesis that accelerates program verification. Our approach employs a principled verification procedure that decomposes the original proof goal into two independent subproblems and launches verifier queries in parallel, providing sound evaluation while achieving state-of-the-art speedup results. Using a benchmark of 866 instances, we evaluated 9 state-of-the-art LLMs across multiple model families and compared against prior LLM-based verifiers. The results demonstrate that Quokka consistently outperforms all existing LLM-based verifiers, achieving speedups of at least 1.2× on 81 instances compared to 39 instances for the previous best approach. We further demonstrated that supervised fine-tuning and Best-of-N sampling can yield measurable improvements in accelerating verification. Our work establishes a new state-of-the-art for LLM-based program verification and provides a principled foundation for future research in this direction.

Chapter 5

Evaluating Program Reasoning via Equivalence Checking

5.1 Introduction

Large language models (LLMs) have rapidly become central to software engineering workflows, powering tools for code generation, program repair, test case generation, debugging, and beyond, significantly boosting developers' productivity [123, 299, 300]. This surge of capability has prompted a natural yet fundamental question: Do LLMs merely mimic code syntax they have seen during training, or do they genuinely understand what programs do?

Unlike natural language, code is executable. Two programs may differ syntactically yet be semantically equivalent, producing identical outputs for all inputs. Conversely, programs with only minor syntactic differences can behave quite differently at runtime. This gap between surface-level program features and actual execution behavior raises an important question: Does training on static code corpora equip LLMs with a grounded understanding of program semantics?

To rigorously assess whether LLMs truly understand code, we need benchmarks that demand reasoning about program semantics. However, widely used coding benchmarks such as HumanEval [34] and MBPP [13] primarily test a model's ability to generate short code snippets from natural language descriptions, offering limited insight into whether the model grasps the underlying semantics of the code it generates.

In this chapter, we introduce **equivalence checking** as a new task for evaluating LLMs' ability to reason about program semantics. Unlike tasks based on syntactic similarity, equivalence checking asks whether two programs are semantically equivalent, i.e., whether they produce identical outputs for all possible inputs, regardless of how differently they are written. Program equivalence problems test directly whether and how well models reason about code. Any question about program semantics

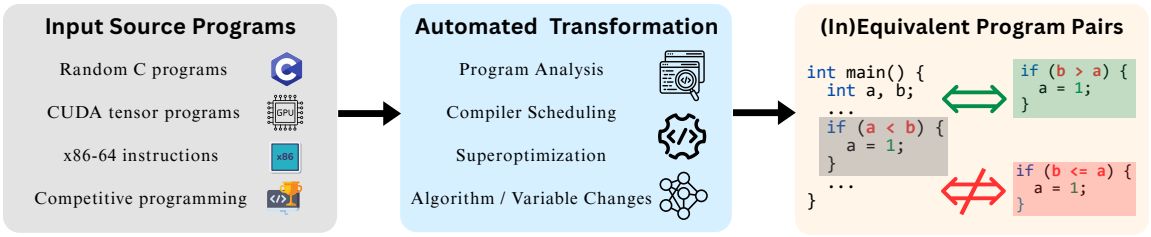


Figure 5.1: Overview of EquiBench. We construct (in)equivalent program pairs from diverse sources, including C and CUDA programs, x86-64 assembly, and competitive programming, using automated transformations based on program analysis, compiler scheduling, superoptimization, and changes in algorithms or variable names.

can be formulated as an equivalence checking problem, and program equivalence problems can have any level of difficulty from trivially easy to extremely difficult. Program equivalence is undecidable in general: no algorithm can determine program equivalence for all cases while guaranteeing termination. This fundamental theoretical impossibility underscores the intrinsic difficulty of our task.

Designing a benchmark for equivalence checking requires both equivalent and inequivalent program pairs spanning diverse categories, which poses several challenges in terms of *label soundness*, *problem difficulty*, and *automation*. First, it is difficult to guarantee high-confidence labels, as verifying equivalence by exhaustively executing all possible inputs is almost always computationally infeasible. Second, existing generation techniques rely on superficial syntactic edits [16, 182], which are too simplistic to meaningfully challenge state-of-the-art LLMs and fail to probe their semantic reasoning limits. Third, to enable comprehensive evaluation, the benchmark must be large-scale and modular, necessitating a fully automated construction pipeline.

In this chapter, we introduce **EquiBench**, a dataset of 2400 program pairs for evaluating large language models on equivalence checking. Covering Python, C, CUDA, and x86-64 programs, it enables a systematic assessment of LLMs' ability to reason about program semantics.

As illustrated in Figure 5.1, EquiBench addresses these challenges by automatically constructing both equivalent and inequivalent program pairs from diverse input sources, including randomly generated C and CUDA code, assembly instructions, and competitive programming solutions. To ensure label soundness without exhaustive execution, we apply program transformation techniques grounded in program analysis and superoptimization. To increase problem difficulty beyond trivial edits, we incorporate structural transformations through compiler scheduling and algorithmic equivalences. The entire generation pipeline is fully automated, enabling scalable construction of a large and diverse benchmark. Finally, EquiBench is extensible to additional categories of equivalence checking problems, which we anticipate will be useful as LLMs improve.

Our experiments show that EquiBench is a challenging benchmark for LLMs. Among the 19 models evaluated, OpenAI o4-mini performs best overall, yet achieves only 60.8% in the CUDA category despite reaching the highest overall accuracy of 82.3%. In the two most difficult categories,

the best accuracies are 63.8% and 76.2%, respectively, only modestly better than the random baseline of 50% for binary classification. In contrast, purely syntactic changes such as variable renaming are much easier, with accuracies as high as 96.5%. We further find, through difficulty analysis, that models often rely on superficial form features such as syntactic similarity rather than demonstrating robust semantic reasoning. Moreover, prompting strategies such as few-shot in-context learning and Chain-of-Thought (CoT) prompting barely improve LLM performance, underscoring the difficulty of reasoning about program semantics.

In summary, our contributions are as follows:

- **New Task and Dataset:** We introduce equivalence checking as a new task to assess LLMs' reasoning about program semantics. We present *EquiBench*, a benchmark for equivalence checking spanning four languages and six equivalence categories.
- **Automated Generation:** We develop a fully automated pipeline for constructing diverse (in)equivalent program pairs using techniques that ensure high-confidence labels and nontrivial difficulty. The pipeline covers transformations ranging from syntactic edits to structural modifications and algorithmic equivalence.
- **Evaluation and Analysis:** We evaluate 19 tate-of-the-art models on EquiBench. In the two most challenging categories, the best accuracies are only 63.8% and 76.2%, highlighting fundamental limitations. Our analysis shows that models often rely on superficial form features rather than demonstrating robust reasoning about program semantics.

5.2 Related Work

LLM Reasoning Benchmarks Extensive research has evaluated LLMs' reasoning capabilities across diverse tasks [45, 118, 26, 188, 321, 111, 272, 33, 44, 309]. In the context of code reasoning, i.e., predicting a program's execution behavior without running it, CRUXEval [94] focuses on input-output prediction, while CodeMind [162] extends evaluation to natural language specifications. Another line of work seeks to improve LLMs' code simulation abilities through prompting [141] or targeted training [163, 192, 61, 35]. Unlike prior work that tests LLMs on specific inputs, our benchmark evaluates their ability to reason over all inputs.

Equivalence Checking Equivalence checking underpins applications such as performance optimization [233, 50, 51], code transpilation [174, 303, 122, 199], refactoring [198], and testing [78, 251]. Due to its undecidable nature, no algorithm can decide program equivalence for all program pairs while always terminating. Existing techniques [228, 54, 101, 189, 42, 15] focus on specific domains, such as SQL query equivalence [312, 60, 236]. EQBENCH [16] and SeqCoBench [182] are the main datasets for equivalence checking, but have limitations. EQBENCH is too small (272 pairs) for

LLM evaluation, while SeqCoBench relies only on statement-level syntactic changes (e.g., renaming variables). In contrast, our work introduces a broader set of equivalence categories, creating a more systematic and challenging benchmark.

5.3 Benchmark Construction

While we have so far discussed the standard notion of equivalence, namely that two programs produce the same output on any input, each benchmark category adopts a more precise definition tailored to its domain. All follow the principle of "producing the same output given the same input," but the exact criteria differ. For example, the CUDA category tolerates small discrepancies from floating-point rounding rather than requiring strict bit-level equivalence. These definitions are grounded in real-world use cases and chosen to capture practical notions of equivalence in each setting. For each category, we provide the corresponding definition in the prompt when testing LLM reasoning. We describe how we generate (in)equivalent pairs across the six categories as follows:

- **DCE**: C program pairs generated via the compiler's dead code elimination (DCE) pass (Section 5.3.1).
- **CUDA**: CUDA program pairs created by applying different scheduling strategies using a tensor compiler (Section 5.3.2).
- **x86-64**: x86-64 assembly program pairs generated by a superoptimizer (Section 5.3.3).
- **OJ_A**, **OJ_V**, **OJ_VA**: Python program pairs from online judge submissions, featuring algorithmic differences (OJ_A), variable-renaming transformations (OJ_V), and combinations of both (OJ_VA) (Section 5.3.4).

5.3.1 Pairs from Program Analysis (DCE)

Dead code elimination (DCE), a compiler pass, removes useless program statements. After DCE, the remaining statements in the modified program naturally *correspond* to those in the original program.

Definition of Equivalence. Two programs are considered equivalent if, when executed on the same input, they *always* have identical *program states* at all corresponding points reachable by program execution. We expect language models to identify differences between the two programs, align their states, and determine whether these states are consistently identical.

Example. Figure 5.2 illustrates an equivalent pair of C programs. In the left program, the condition (`p1 == p2`) compares the memory address of the first element of the array `b` with that of the static variable `c`. Since `b` and `c` reside in different memory locations, this condition can never be satisfied.

```
char b[2];
static int c = 0;

int main() {
  char* p1 = &b[0];
  int*  p2 = &c;
  ...
  if (p1 == p2) {
     // dead code
     c = 1;
  }
  return 0;
}
```

```
char b[2];
static int c = 0;

int main() {
  char* p1 = &b[0];
  int*  p2 = &c;
  ...
  if (p1 == p2) {
   // code eliminated
  }
  return 0;
}
```

Figure 5.2: An equivalent pair from the DCE category in EquiBench. In the left program, `c = 1` is dead code that has no effect on the program state and is removed in the right program. Such pairs are generated using the Dead Code Elimination (DCE) pass in compilers.

```
__global__ void GEMV(const float* A,
                     const float* x,
                     float* y,
                     int R,
                     int C) {

    // Calculate the row index
    // assigned to the thread
    int r = blockIdx.x * blockDim.x
            + threadIdx.x;

    // Return if out of bounds
    if (r >= R) return;
    float s = 0.0f;

    for (int c = 0; c < C; c++) {
        s += A[r * C + c] * x[c];
    }

    y[r] = s;
}
```

```
__global__ void GEMV(const float* A, const float* x,
                     float* y, int R, int C) {
    __shared__ float tile[32]; // tiling with shared memory
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    bool valid = (r < R);
    float s = 0.0f;
    for (int start = 0; start < C; start += 32) {
        for (int i = threadIdx.x; i < 32; i += blockDim.x) {
            int c = start + i;
            if (c < C) tile[i] = x[c]; // load x into tile
        }
        __syncthreads();
        if (valid) {
            for (int j = 0; j < min(32, C - start); j++) {
                s += A[r * C + (start + j)] * tile[j];
            }
        }
        __syncthreads();
    }
    if (valid) y[r] = s;
}
```

Figure 5.3: An equivalent pair from the CUDA category in EquiBench. Both programs perform matrix-vector multiplication ($y = Ax$). The right-hand program uses *shared memory tiling* to improve performance. Tensor compilers are utilized to explore different *scheduling strategies*, automating the generation.

As a result, the assignment `c = 1` is never executed in the left program and is removed in the right program.

Automation. This reasoning process is automated by compilers through *alias analysis*, which statically determines whether two pointers can reference the same memory location. Based on this analysis, the compiler's *Dead Code Elimination (DCE)* pass removes code that does not affect

program semantics to improve performance.

Dataset Generation. We utilize CSmith [302] to create an initial pool of random C programs. Building on techniques from prior compiler testing research [250], we implement an LLVM-based tool [145] to classify code snippets as either dead or live. Live code is further confirmed by executing random inputs with observable side effects. Equivalent program pairs are generated by eliminating dead code, while inequivalent pairs are generated by removing live code.

5.3.2 Pairs from Compiler Scheduling (CUDA)

Definition of Equivalence. Two CUDA programs are considered equivalent if they produce the same mathematical output for any valid input, *disregarding floating-point rounding errors*. This definition *differs* from that in Section 5.3.1, as it does not require the internal program states to be identical during execution.

Example. Figure 5.3 shows an equivalent CUDA program pair. Both compute matrix-vector multiplication $y = Ax$, where A has dimensions (R, C) and x has size C. The right-hand program applies the *shared memory tiling* technique, loading `x` into shared memory `tile` (declared with `__shared__`). Synchronization primitives `__syncthreads()` are properly inserted to prevent synchronization issues.

Automation. The program transformation can be automated with tensor compilers, which provide a set of *schedules* to optimize loop-based programs. These schedules include loop tiling, loop fusion, loop reordering, loop unrolling, vectorization, and cache optimization. For any given schedule, the compiler can generate the transformed code. While different schedules can significantly impact program performance on the GPU, they do not affect the program's correctness (assuming no compiler bugs), providing the foundation for automation.

Dataset Generation. We utilize TVM as the tensor compiler [36] and sample tensor program schedules from TenSet [316] to generate equivalent CUDA program pairs. Inequivalent pairs are created by sampling code from different tensor programs.

5.3.3 Pairs from a Superoptimizer (x86-64)

Definition of Equivalence. Two x86-64 assembly programs are considered equivalent if, for any input provided in the specified input registers, both programs produce identical outputs in the specified output registers. Differences in other registers or memory are ignored for equivalence checking.

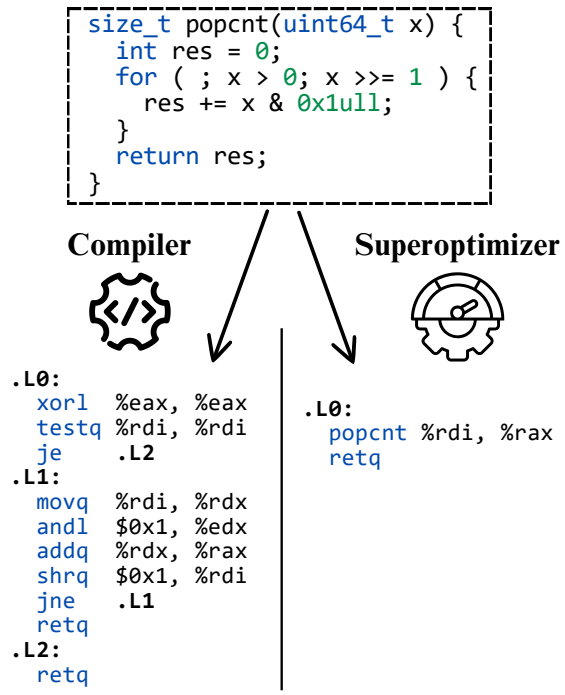


Figure 5.4: An equivalent pair from the x86-64 category in EquiBench. Both programs are compiled from the same C function shown above, the left using a compiler and the right using a *superoptimizer*. The function counts the number of set bits in the input `%rdi` register and stores the result in `%rax`. Their equivalence has been formally verified by the superoptimizer.

Example. Figure 5.4 shows an example of an equivalent program pair in x86-64 assembly. Both programs implement the same C function, which counts the number of bits set to 1 in the variable `x` (mapped to the `%rdi` register) and stores the result in `%rax`. The left-hand program, generated by GCC with O3 optimization, uses a loop to count each bit individually, while the right-hand program, produced by a superoptimizer, leverages the `popcnt` instruction, a hardware-supported operation for efficient bit counting. The superoptimizer verifies that both programs are semantically equivalent. Determining this equivalence requires a solid understanding of x86-64 assembly semantics and the ability to reason about all possible bit patterns.

Automation. A superoptimizer searches a space of programs to find one equivalent to the target. Test cases efficiently prune incorrect candidates, while formal verification guarantees the correctness of the optimized program. Superoptimizers apply aggressive and non-local transformations, making semantic equivalence reasoning more challenging. For example, in Figure 5.4, while a traditional compiler translates the loop in the source C program into a loop in assembly, a superoptimizer can find a more optimal instruction sequence by leveraging specialized hardware instructions. Such transformations are beyond the scope of traditional compilers.

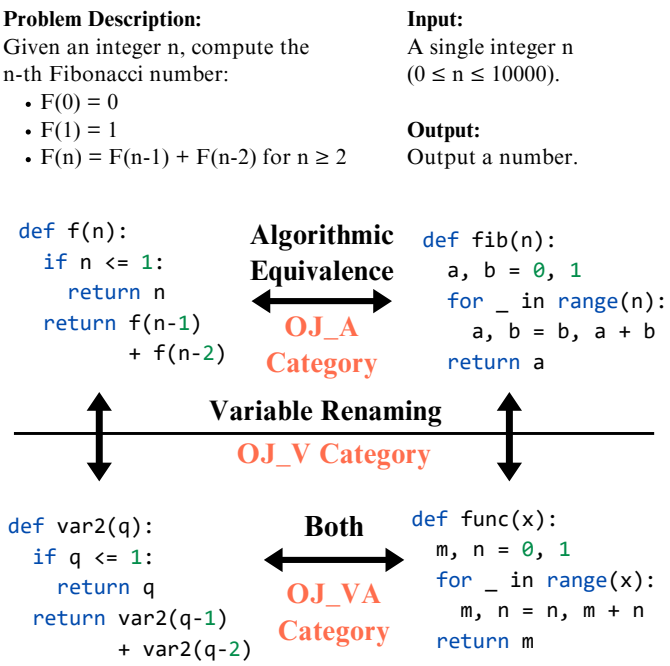


Figure 5.5: Equivalent pairs from the OJ_A, OJ_V, OJ_VA categories in EquiBench. OJ_A pairs demonstrate *algorithmic equivalence*, OJ_V pairs involve *variable renaming* transformations, and OJ_VA pairs combine *both* types of variations.

Dataset Generation. We use Stoke [216] to generate program pairs. Assembly programs are sampled from prior work [137], and Stoke applies transformations to produce candidate programs. If verification succeeds, the pair is labeled as equivalent; if the generated test cases fail, it is labeled as inequivalent.

5.3.4 Pairs from Programming Contests

Definition of Equivalence. Two programs are considered equivalent if they solve the same problem by producing the same output for any valid input, as defined by the problem description. Both programs, along with the problem description, are provided to determine equivalence.

Example. Given the problem description in Figure 5.5, all four programs are equivalent as they correctly compute the nth Fibonacci number. The **OJ_A** pairs demonstrate **algorithmic** equivalence—the left-hand program uses recursion, while the right-hand program employs a for-loop. The **OJ_V** pairs are generated through **variable renaming**, a purely syntactic transformation that obscures the program's semantics by removing meaningful variable names. The **OJ_VA** pairs combine **both** algorithmic differences and variable renaming.

Category	Language	# Pairs	Lines of Code		
			Min	Max	Avg.
DCE	C	400	98	880	541
CUDA	CUDA	400	46	1733	437
x86-64	x86-64	400	8	29	14
OJ_A	Python	400	3	3403	82
OJ_V	Python	400	2	4087	70
OJ_VA	Python	400	3	744	35

Table 5.1: Statistics of the EquiBench dataset.

Dataset Generation. We sample Python submissions using a publicly available dataset from Online Judge (OJ) [208]. For OJ_A pairs, accepted submissions are treated as equivalent, while pairs consisting of an accepted submission and a wrong-answer submission are considered inequivalent. Variable renaming transformations are automated with an open-source tool [86].

5.4 Experimental Setup

Dataset. EquiBench consists of 2,400 program pairs across six equivalence categories, each with 200 equivalent and 200 inequivalent pairs. Table 5.1 summarizes the statistics of program lengths. Constructing the program pairs required substantial systems effort. For example, for the DCE category, we developed a 2,917-line LLVM-based tool, including 1,472 lines in C and C++, with alias analysis and path feasibility analysis to accurately classify live and dead code.

Prompts. The 0-shot evaluation is conducted using the prompt "You are here to judge if two programs are semantically equivalent. Here equivalence means {*definition*}. [Program 1]: {code1} [Program 2]: {code2} Please only output the answer of whether the two programs are equivalent or not. You should only output Yes or No." The definition of equivalence and the corresponding program pairs are provided for each category. Additionally, for the categories of OJ_A, OJ_V, and OJ_VA, the prompt also includes the problem description.

5.5 Results

5.5.1 Model Accuracy

Table 5.2 shows the accuracy results for 19 state-of-the-art large language models on EquiBench under zero-shot prompting. Our findings are as follows:

Reasoning models achieve the highest performance. As shown in Table 5.2, reasoning models such as OpenAI o3-mini, DeepSeek R1, and o1-mini significantly outperform all others in our

Model	DCE	CUDA	x86-64	OJ_A	OJ_V	OJ_VA	Overall
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*
Llama-3.2-3B-Instruct-Turbo	50.0	49.8	50.0	51.5	51.5	51.5	50.7
Llama-3.1-8B-Instruct-Turbo	41.8	49.8	50.5	57.5	75.5	56.8	55.3
Mistral-7B-Instruct-v0.3	51.0	57.2	73.8	50.7	50.5	50.2	55.6
Mixtral-8x7B-Instruct-v0.1	50.2	47.0	64.2	59.0	61.5	55.0	56.1
Mixtral-8x22B-Instruct-v0.1	46.8	49.0	62.7	63.5	76.0	62.7	60.1
Llama-3.1-70B-Instruct-Turbo	47.5	50.0	58.5	66.2	72.0	67.5	60.3
QwQ-32B-Preview	48.2	50.5	62.7	65.2	71.2	64.2	60.3
Qwen2.5-7B-Instruct-Turbo	50.5	49.2	58.0	62.0	80.8	63.0	60.6
gpt-4o-mini-2024-07-18	46.8	50.2	56.8	64.5	91.2	64.0	62.2
Qwen2.5-72B-Instruct-Turbo	42.8	56.0	64.8	72.0	76.5	70.8	63.8
Llama-3.1-405B-Instruct-Turbo	40.0	49.0	75.0	72.2	74.5	72.8	63.9
DeepSeek-V3	41.0	50.7	69.2	73.0	83.5	72.5	65.0
gpt-4o-2024-11-20	43.2	49.5	65.2	71.0	87.0	73.8	65.0
claude3.5-sonnet-2024-10-22	38.5	62.3	70.0	71.2	78.0	73.5	65.6
claude3.7-sonnet-2025-04-16	40.5	**63.8**	64.8	70.5	89.2	73.5	67.0
o1-mini-2024-09-12	55.8	50.7	74.2	80.0	89.8	78.8	71.5
DeepSeek-R1	52.2	61.0	78.2	79.8	91.5	78.0	73.5
o3-mini-2025-01-31	68.8	59.0	**84.5**	84.2	88.2	83.2	78.0
o4-mini-2025-04-16	**76.2**	60.8	83.0	**89.0**	**96.5**	**88.5**	**82.3**
Mean	49.0	53.4	66.7	68.6	78.1	68.5	64.0

Table 5.2: Accuracy of 19 models on EquiBench under 0-shot prompting. We report accuracy for each of the six equivalence categories along with the overall accuracy.

evaluation. This further underscores the complexity of equivalence checking, where reasoning models exhibit a distinct advantage.

EquiBench is a challenging benchmark. Among the 19 models evaluated, OpenAI o4-mini achieves only 60.8% in the CUDA category despite being the top-performing model overall, with an accuracy of 82.3%. For the two most difficult categories, the highest accuracy across all models is 63.8% and 76.2%, respectively, only modestly above the random baseline of 50% accuracy for binary classification, highlighting the substantial room for improvement.

Scaling up models improves performance. Larger models generally achieve better performance. Figure 5.6 shows scaling trends for the Qwen2.5, Llama-3.1, and Mixtral families, where accuracy improves with model size. The x-axis is on a logarithmic scale, highlighting how models exhibit consistent gains as parameters increase.

5.5.2 Difficulty Analysis

We conduct a detailed difficulty analysis across equivalence categories and study how syntactic similarity influences model predictions.

Difficulty by Transformation Type. Each category adopts a specific definition of equivalence (see Section 5.3), and the program transformations used in each category differ accordingly. We find that purely syntactic transformations are substantially easier for models, while structural and compiler-involved transformations are much harder. Specifically, **OJ_V** (variable renaming) achieves the highest mean accuracy of 78.1%, as it only requires surface-level reasoning. **OJ_A** (algorithmic equivalence) and **OJ_VA** (variable renaming combined with algorithmic differences) achieve similar accuracies of 68.6% and 68.5%, respectively. Additionally, for the **OJ_A** category, since we do not impose strict constraints on algorithmic differences between program pairs, it may also make the task easier for models if the two submissions are close. In contrast, **x86-64** (66.7%) and **CUDA** (53.4%) involve complex instruction-level or memory-level transformations, requiring deeper semantic reasoning. **DCE** (dead code elimination) is the most difficult category, with a mean accuracy of 49.0%, suggesting that models struggle with nuanced program analysis concepts. We further discuss the possibility of dataset contamination below, which may also affect model accuracy across different categories.

Possibility of Dataset Contamination. We observe that the categories **OJ_V** (variable renaming), **OJ_A** (algorithmic equivalence), and **OJ_VA** (variable renaming combined with algorithmic differences) achieve the highest accuracy. The programs in these categories are sourced from programming contests via the CodeNet dataset [208]. Since CodeNet is designed to help AI systems learn code understanding and improvement, it is likely that these programs appear in the training data of many LLMs, which may contribute to their strong performance on these categories.

Difficulty by Syntactic Similarity. To assess whether LLM predictions reflect understanding of program semantics rather than reliance on surface-level syntax, we analyze how syntactic similarity affects model behavior. Using Moss [217], a plagiarism detection tool, we observe the following:

- For program pairs with low syntactic similarity, models tend to predict "inequivalent," even when the programs are semantically equivalent. This suggests an overreliance on the superficial form of the code.
- For syntactically similar pairs, models are more likely to predict "equivalent," indicating a tendency to associate similarity in form with equivalence in program semantics.

We validate this trend through statistical testing: at significance level ($\alpha = 0.05$), model accuracy on equivalent pairs increases with syntactic similarity, while accuracy on inequivalent pairs decreases. This disconnect between syntactic form and execution behavior, as discussed in Section 5.1, suggests that models do not fully grasp program semantics.

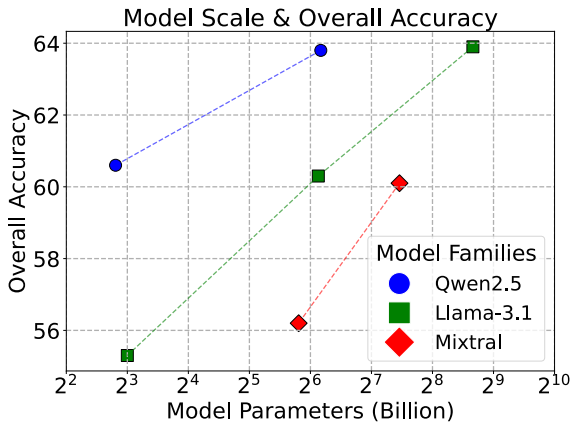


Figure 5.6: Scaling Trend on EquiBench. Models exhibit consistent gains as parameters increase.

Implications for Benchmark Design. These findings suggest that future benchmarks should emphasize *syntactically dissimilar yet equivalent program pairs* and *syntactically similar yet inequivalent program pairs* to create more challenging and diagnostic benchmarks for evaluating the deep semantic reasoning capabilities of LLMs.

5.5.3 Bias in Model Prediction

We evaluate the prediction bias of the models and observe a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories. Table 5.3 presents the results for four representative models, showing high accuracy for inequivalent pairs but significantly lower accuracy for equivalent pairs.

The bias in the CUDA category arises from extensive structural transformations, such as loop restructuring and shared memory optimizations, which make paired programs appear substantially different. In the x86-64 category, superoptimization applies non-local transformations to achieve optimal instruction sequences, introducing aggressive code restructuring that complicates equivalence reasoning and leads models to misclassify equivalent pairs as inequivalent frequently.

5.5.4 Prompting Strategies Analysis

We study few-shot in-context learning and Chain-of-Thought (CoT) prompting, evaluating four strategies: 0-shot, 4-shot, 0-shot with CoT, and 4-shot with CoT. For 4-shot, prompts include 2 equivalent and 2 inequivalent pairs. Table 5.4 shows the results.

Our key finding is that prompting strategies barely improve performance on EquiBench, highlighting the difficulty of understanding program semantics.

Model	CUDA		x86-64	
	Eq	**Ineq**	**Eq**	**Ineq**
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
o3-mini	27.5	90.5	69.5	99.5
o1-mini	2.5	99.0	50.0	98.5
DeepSeek-R1	28.0	94.0	57.5	99.0
DeepSeek-V3	8.5	93.0	44.0	94.5

Table 5.3: Accuracies on equivalent and inequivalent pairs in the CUDA and x86-64 categories under 0-shot prompting, showing that models perform significantly better on inequivalent pairs. Random guessing serves as an unbiased baseline for comparison.

Model	**0S**	**4S**	**0S-CoT**	**4S-CoT**
o1-mini	71.5	71.5	**71.9**	**71.9**
gpt-4o	65.0	**66.5**	62.5	62.7
DeepSeek-V3	65.0	**66.9**	63.3	62.5
gpt-4o-mini	62.2	**63.5**	60.2	61.2

Table 5.4: Accuracies of different prompting techniques. We evaluate 0-shot and 4-shot in-context learning, both without and with Chain-of-Thought (CoT). Prompting strategies barely improve performance.

5.6 Discussion and Future Directions

Scope and Positioning Machine learning has been applied to many code-related tasks, such as clone detection [274], code search [90], and bug finding [58]. EquiBench focuses on equivalence checking, which differs fundamentally by evaluating a model's understanding of program semantics. Unlike natural language, code is executable, and its correctness depends on execution results rather than form. For example, clone detection captures syntactic or structural similarity without considering behavior. In contrast, EquiBench tests whether two programs produce the same outputs for all inputs, offering an informative benchmark for reasoning about program behavior.

Developer Use Cases EquiBench evaluates whether LLMs truly understand program semantics, a capability that underpins downstream tasks such as program optimization, software refactoring, and transpilation. These tasks are central to practical scenarios where coding assistants must propose improvements or transformations without changing program behavior. For example, after a developer performs a refactoring, a coding assistant that performs well on EquiBench would be better positioned to judge whether the transformed code preserves the same functionality as the original.

Labeling Soundness To ensure high-assurance equivalence labels, EquiBench relies on transformations grounded in program analysis, compiler scheduling, and superoptimization, all of which offer strong soundness guarantees. In contrast, approaches such as random testing [127], similarity-based

tools [235], and refactoring datasets lack formal guarantees and risk introducing incorrect labels.

Design and Extensibility EquiBench is designed with modularity in mind: each equivalence category corresponds to a distinct class of program transformations. Demonstrating strong performance in these settings would indicate that LLMs could support some components of compiler pipelines (e.g., dead code elimination (DCE) as a core compiler optimization, or CUDA program scheduling for high-performance ML systems). We focus on the six categories where large-scale, high-confidence labels can be generated automatically. That said, equivalence checking is a general task that is applicable to all programming languages. We view our benchmark as a first step, and its modular design allows future extension to additional categories and languages.

Evaluation of Reasoning Trace While our evaluation centers on binary classification, understanding the rationale behind model predictions is an important direction. Explanations may take the form of natural language or formal proofs, but verifying their correctness remains difficult. Natural language lacks reliable automated validation, since using LLMs as judges can produce unsound results. Building a proof-based evaluation framework using tools such as Lean is also highly nontrivial. We present a manual case analysis of reasoning trace correctness in Section 6.5.5 and leave automated robust evaluation of reasoning as future work.

Effect of Fine-Tuning We tested whether supervised fine-tuning improves performance. Fine-tuning Qwen2.5-14B-Instruct with LoRA for 3 epochs on 1,200 labeled examples increased accuracy from 59.8% to 63.2%. The small gain suggests that binary labels alone provide limited learning signals for reasoning about program semantics. Prior work has explored training with *program execution traces* to better capture execution behavior. We conducted an additional experiment to evaluate the training approach from SemCoder [61]. The base model (DeepSeek-Coder-6.7B) achieves 49.9% accuracy on our benchmark, and the fine-tuned model released by SemCoder reaches 54.9%. While this shows some benefit, the improvement remains modest. These results support our broader claim: EquiBench presents a difficult and meaningful challenge even for fine-tuned models, and deeper semantic understanding remains out of reach for current approaches.

Future Directions We believe EquiBench can inform future research on task-specific training methods, including: (1) distilling reasoning traces from stronger models, (2) scaling training with larger datasets generated through our pipeline, (3) developing agentic approaches where LLMs actively execute and compare programs using tools (e.g., a Python interpreter) to generate inputs that expose differences, (4) applying reinforcement learning with execution-based feedback, and (5) creating datasets with program analysis concepts (see Section 6.5.5) for training LLMs.

5.7 Conclusion

EquiBench is a benchmark for evaluating whether large language models (LLMs) truly understand program semantics. We propose the task of equivalence checking, which asks whether two programs produce identical outputs for all possible inputs, as a direct way to test a model's ability to reason about program behavior. The dataset consists of 2400 program pairs across four languages and six categories, constructed through a fully automated pipeline that provides high-confidence labels and nontrivial difficulty. Our evaluation of 19 state-of-the-art LLMs shows that even the best-performing models achieve only modest accuracy in the most challenging categories. Further analysis shows that LLMs often rely on syntactic similarity instead of demonstrating robust reasoning about program semantics, underscoring the need for further advances in the semantic understanding of programs.

Chapter 6

Evaluating Inductive Reasoning via Program Synthesis

6.1 Introduction

Inductive reasoning, i.e., the ability to identify patterns and form abstractions from limited examples, is widely recognized as a fundamental aspect of human intelligence [108, 294]. In the context of programming, inductive reasoning underpins the task of synthesizing functions that satisfy given input-output examples and generalize to unseen inputs. This task, commonly referred to as *inductive program synthesis* or programming by example [180, 191, 83, 155], has broad application domains [259, 56].

Recent advances in Large Language Models (LLMs) have led to the emergence of autonomous agents capable of decision-making, multi-step planning, tool use, and iterative self-improvement through interaction and feedback [38, 172, 170, 99, 282]. While much of the existing work focuses on programming tasks guided by natural language [34, 13, 128, 124], we study a fundamentally different problem: inductive program synthesis, where the objective is to infer the target program solely from input-output examples. This setting provides a more rigorous test of inductive reasoning capabilities, as it eliminates natural language descriptions that can trigger retrieval-based completions memorized during model training.

Designing an effective evaluation protocol for inductive program synthesis with LLMs is inherently challenging, as multiple valid functions may satisfy a given set of input-output examples (as demonstrated in Section 6.5.2). The state-of-the-art protocol [155], which evaluates synthesized functions on held-out test cases after presenting 10 fixed input-output examples, has several notable limitations. First, a small, static set of input-output examples may underspecify the target functions, especially for those with complex logic. Moreover, held-out tests may fail to reveal subtle semantic

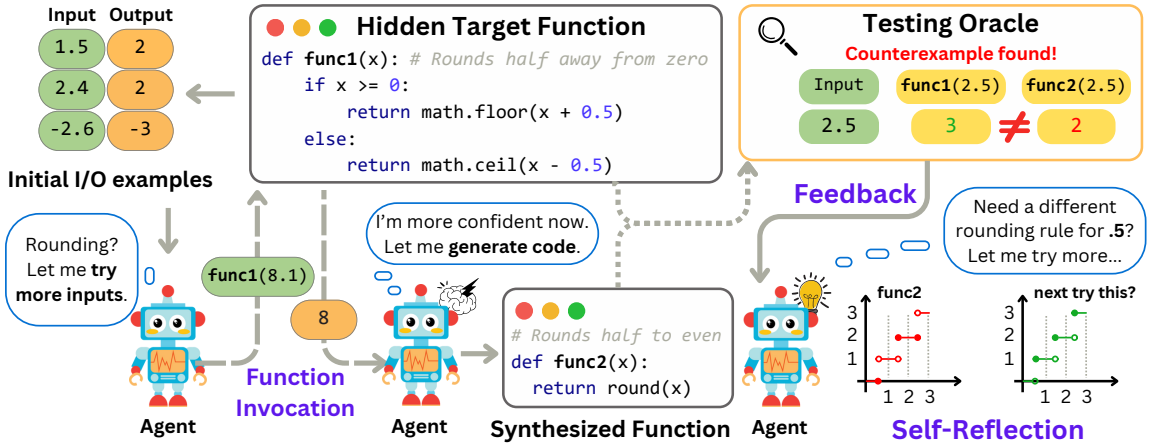


Figure 6.1: Overview of CodeARC. Our framework evaluates LLMs' reasoning capabilities in inductive program synthesis. The agent begins with input-output examples, interacts with a hidden target function via function calls, and uses a differential testing oracle to check the correctness of the synthesized function for self-reflection and refinement.

discrepancies between the generated and intended implementations. In addition, when the model produces an incorrect solution, it receives no feedback and has no opportunity to revise or explore alternatives. Finally, existing benchmarks for inductive program synthesis [95, 259, 276, 56, 155] are focused on domain-specific tasks and do not assess the ability of LLMs to synthesize functions written in general-purpose programming languages.

To address the limitations of existing evaluation protocols for inductive program synthesis, we introduce CodeARC, the *Code Abstraction and Reasoning Challenge*, inspired by real-world scenarios such as decompilation and reverse engineering [159, 181, 8]. In such settings, an agent is given a binary executable (without source code) and must synthesize equivalent source code by observing input-output behavior. Instead of relying on a fixed dataset, the agent can query the binary with new inputs, invoke a differential testing oracle, and use counterexamples for iterative refinement. This setup parallels the classic learnability framework of queries and counterexamples [9, 55], here applied to program synthesis.

Figure 6.1 illustrates how CodeARC instantiates this process: LLM-based agents begin with an initial set of input-output examples, query the ground-truth function for more examples, and debug synthesized code using a differential testing oracle. We impose fixed budgets on both the number of observable input-output examples and the number of testing oracle invocations for self-debugging. The task requires agents to proactively generate inputs (function calls) and revise solutions based on feedback (self-reflection). This interactive setup offers a more realistic alternative to prior static evaluation protocols.

We construct the first comprehensive dataset for general-purpose inductive program synthesis,

featuring 1114 functions with initial input-output examples. Our benchmark targets general programming tasks and employs two state-of-the-art differential testing tools [177, 71] for correctness evaluation.

Our experiments demonstrate that CodeARC poses a significant challenge for LLM-based inductive program synthesis. Among the 18 models evaluated, OpenAI o3-mini performs the best overall, yet only achieves 52.7% success rate. We further conduct ablation studies to analyze how budgets on the number of input-output examples and oracle calls affect model performance. To enhance model capabilities, we generate synthetic fine-tuning data with curated synthesis traces that capture the reasoning steps. We show that supervised fine-tuning on LLaMA-3.1-8B-Instruct yields up to a 31% relative performance improvement.

In summary, our contributions are as follows:

- **Interactive evaluation protocol for inductive program synthesis.** We introduce a setup where agents start with fixed input-output examples but can generate new inputs to query ground-truth functions and invoke a differential testing oracle to self-correct their solutions. This setup brings the task closer to a real-world setting, e.g. reverse-engineering.
- **General-purpose benchmark with extensive evaluation.** We construct the first large-scale, general-purpose benchmark for this task, including 1114 diverse functions. Among the 18 models evaluated, o3-mini achieves the best overall performance but still only reaches a success rate of 52.7%.
- **Synthetic data and fine-tuning.** To boost model performance, we generate synthetic fine-tuning data containing curated synthesis traces that capture both function invocations and reasoning steps. We show that fine-tunining on LLaMA-3.1-8B-Instruct yields up to a 31% relative performance improvement.

6.2 Related Work

Inductive Program Synthesis Traditional inductive program synthesis methods rely solely on input-output examples, without natural language input. They focus on domain-specific tasks like string and data transformations [95, 237, 292], SQL [259], visual programming [260], and quantum computing [56]. These approaches typically define a domain-specific language and use tailored search algorithms to prune the space efficiently [84, 207, 81, 103, 184]. In contrast, we introduce the first general-purpose program synthesis benchmark for LLM-powered agents.

LLM Benchmarks for Code Most LLM benchmarks, such as HumanEval+ [34, 167], MBPP+ [13, 167], APPS [110], and others [157, 168, 154, 123, 17, 193, 171, 65, 243, 323, 306, 143, 202, 254, 253, 270, 279], evaluate code generation from natural language. Beyond generation, tasks like I/O

prediction [94, 162], execution prediction [163, 141, 192, 61], bug localization [242], and program equivalence [264] have also been studied. In contrast, we focus on predicting function bodies purely from input-output examples, without natural language. Prior work [155, 19] targets domain-specific tasks, while we introduce a general-purpose benchmark with an interactive evaluation protocol.

LLM Benchmarks for Reasoning LLMs are widely benchmarked on reasoning tasks across domains, including commonsense [246], mathematical [45], and logical [106, 185, 164, 200, 271]. Inductive reasoning, a core cognitive skill that generalizes from limited examples [108], is increasingly studied in LLMs [150, 178, 288, 28, 224]. ARC [41] is a prominent benchmark for abstract pattern induction. Our work shares this goal but is for inductive program synthesis.

LLM-powered Agents LLM-based agents have shown strong performance in domains like web navigation [320, 322], code generation [310, 129], performance optimization [265, 269], and ML experimentation [119]. They interact with environments, invoke functions, make decisions, and self-reflect [179, 203, 289, 117, 231]. We introduce the first benchmark to systematically evaluate agents' capabilities in inductive program synthesis, providing a rigorous testbed for inductive reasoning and program synthesis.

6.3 Method

6.3.1 Problem Definition of Inductive Program Synthesis

We formalize the inductive program synthesis task as follows. Let f^* be a *hidden* ground-truth function that maps inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}$. The synthesizer is given an initial set of input-output examples $\mathcal{E}_0 = \{(x_i, y_i)\}_{i=1}^n$, where $y_i = f^*(x_i)$, and the goal is to synthesize a program $\hat{f}$ such that $\hat{f} \equiv f^*$, i.e.,

$$\forall x \in \mathcal{X}, \quad \hat{f}(x) = f^*(x).$$

To evaluate whether a synthesized function $\hat{f}$ is correct, we introduce a *differential testing oracle* $\mathcal{O}$. The oracle takes as input both the synthesized function $\hat{f}$ and the hidden ground-truth function f^* and attempts to identify inputs on which their behaviors differ. Formally, the oracle operates as follows:

$$\mathcal{O}(f^*, \hat{f}) = \begin{cases} \text{Pass}, & \text{if } \forall x \in \mathcal{X}_{\text{test}},\ \hat{f}(x) = f^*(x); \\ \text{Fail}(x), & \text{if } \exists x \in \mathcal{X}_{\text{test}} \text{ such that } \hat{f}(x) \neq f^*(x), \end{cases}$$

where $\mathcal{X}_{\text{test}} \subseteq \mathcal{X}$ is a set of test inputs dynamically selected by the oracle.

Unlike fixed held-out test sets used in prior work, the differential testing oracle conditions on both f^* and the candidate $\hat{f}$, generating targeted inputs to expose discrepancies. On failure, it returns a counterexample $x \in \mathcal{X}_{\text{test}}$ such that $\hat{f}(x) \neq f^*(x)$. Note that program equivalence checking is

fundamentally undecidable, and thus no perfect oracle exists. To approximate oracle functionality, we adopt two state-of-the-art differential testing tools, enabling a more robust and practical evaluation than prior work or reliance on a single tool.

6.3.2 Interactive Evaluation Protocol for LLM Agents

To evaluate the capabilities of LLM-based agents in inductive program synthesis, we introduce an interactive protocol. This protocol extends beyond static evaluation settings by enabling dynamic interaction with the hidden ground-truth function and the differential testing oracle.

Initial Information. At the beginning of the task, the agent is provided with an initial set of input-output examples $\mathcal{E}_0 = \{(x_i, y_i)\}_{i=1}^{n}$, as explained previously. This set serves as partial information about the target function.

Action Space. During evaluation, the agent may take two types of actions. First, it may query the ground-truth function f^* at a chosen input $x \in \mathcal{X}$, and obtain the corresponding output $f^*(x)$, thereby augmenting its observed set of input-output pairs. Second, it may synthesize a candidate program $\hat{f}$ and invoke the differential testing oracle $\mathcal{O}(f^*, \hat{f})$, which returns PASS if no discrepancies are found on a dynamically generated test set, or FAIL with a counterexample $x \in \mathcal{X}_{\text{test}}$ such that $\hat{f}(x) \neq f^*(x)$.

Self-Reflection. If the oracle returns FAIL, a counterexample δ, which is a tuple of $(x, \hat{f}(x), f^*(x))$ will be provided to the agent. This counterexample helps the agent to self-reflect and revise its current hypothesis, either by issuing additional queries f^* or synthesizing new programs. The ability to take such feedback is crucial for iterative refinement.

Budget Constraints. The agent operates under two budget parameters: B_{io} and B_{oracle}. B_{io} limits the total number of input-output examples that the agent can observe from the ground-truth function f^*, while B_{oracle} limits the number of invocations to the differential testing oracle $\mathcal{O}$.

Evaluation Metrics. The task is considered successful if the final synthesized program $\hat{f}$, produced within the given budgets, receives a PASS from the differential testing oracle $\mathcal{O}(f^*, \hat{f})$. We assess LLM agent performance along two dimensions: correctness and efficiency, prioritizing correctness. Correctness is the success rate, i.e., the proportion of problems solved within the budget. Efficiency is the average number of input-output queries and oracle invocations per problem.

6.3.3 Benchmark Preparation

Our benchmark is designed to evaluate LLM agents on the inductive synthesis of general-purpose Python programs. This contrasts with prior benchmarks that focus on domain-specific tasks or programs written in domain-specific languages, such as string manipulation and SQL query generation [293, 56, 155].

Programs for Synthesis. We curate a diverse collection of Python functions sampled from three established code generation benchmarks: HumanEval [34], MBPP [13], and APPS [110]. HumanEval and MBPP primarily consist of simple, entry-level programming tasks, whereas APPS contains more challenging problems that resemble competition-level code exercises. Importantly, we extract only the function bodies from these benchmarks and do not use any accompanying natural language descriptions.

Annotated vs. Anonymized. To assess the extent to which function names help the LLM agent synthesize the correct program, we construct two versions of the benchmark. In the *annotated* version, function names that reflect the intended functionality of the task (e.g., `is_palindrome`) are made available to the agent. In the *anonymized* version, all function names are replaced with a generic identifier (i.e., `solution`). This design isolates the influence of identifier cues on synthesis performance. We report results on both versions.

Initial Input-Output Examples. Each synthesis task includes 10 fixed input-output examples that specify the target function's expected behavior. We use GPT-4o to generate diverse inputs and execute the original function to obtain the corresponding ground-truth outputs.

Synthetic Data Generation. We generate a synthetic dataset for fine-tuning (described in more detail in Section 6.3.4) by first collecting 50 seed Python functions that are *disjoint* from the evaluation set. Using these seeds, we prompt GPT-4o to synthesize a diverse set of new functions, yielding 10,000 candidates. For each generated function, we additionally instruct GPT-4o to produce 10 representative inputs that expose the function's behavior and highlight patterns in its input-output relationships. These inputs are executed to verify the executability of the functions, and we discard any that fail at this stage. To reduce redundancy, we deduplicate the dataset based on function names, finally resulting in 5,405 unique Python functions used for fine-tuning.

6.3.4 Fine-Tuning on Synthetic Data

We evaluate whether fine-tuning on curated synthesis traces, which capture both function calls and reasoning, improves LLM performance on CodeARC, using a distillation approach that imitates the reasoning of a teacher model with access to f^*, the ground-truth function.

During training, we first run the interactive evaluation protocol described in Section 6.3.2 with a frozen teacher model. Unlike a standard evaluation, we prepend a set of task-specific instructions P_{f^*} to each teacher prompt. This prefix includes the function body of f^* and explicitly instructs the teacher to (1) query f^* on informative inputs, (2) explain the rationale behind those queries, and (3) synthesize the full implementation of f^* only when confident that the correct logic can be inferred from the accumulated input-output pairs. The student model, which is the only model we fine-tune, learns to mimic the teacher's reasoning and synthesis behavior without seeing the teacher's prompt that includes the target function.

We provide the teacher with access to f^* because we find that, in many cases, the model struggles to solve the task independently. Without knowledge of the ground truth, the teacher is often too weak to generate meaningful queries or explanations, limiting the effectiveness of the resulting supervision.

While executing the evaluation protocol, we record the multi-turn conversation history C_T, which comprises the teacher model's prompts and responses. Let n be the total number of turns in C_T and denote by x^i the sequence of tokens in the ith turn. Furthermore, let p represent the number of tokens in the teacher-specific instruction prefix P_{f^*}.

We fine-tune the student model using a language modeling objective by minimizing the negative log-likelihood of predicting the next token in C_T:

$$\mathcal{L} = -\sum_{i=1}^{n} \sum_{j=p+1}^{|x^i|} \log P\left(x_j^i \mid C_{T,<i}, x_{<j}^i\right).$$

Here, $P\left(x_j^i \mid C_{T,<i}, x_{<j}^i\right)$ denotes the probability of generating token x_j^i given all tokens from previous turns, $C_{T,<i}$, and the tokens preceding x_j^i in the current turn, $x_{<j}^i$. By starting the inner sum at $j = p + 1$, the teacher-specific instructions P_{f^*} are excluded from the loss computation, ensuring that the training signal comes only from the parts of C_T available during inference.

6.4 Experiment Setup

We construct two versions (annotated and anonymized) of the 1114 Python functions, drawn from HumanEval+, MBPP+ [167], and APPS [110]. Table 6.1 summarizes key statistics. Unlike prior work in program synthesis that often focuses on domain specific languages and constrained settings, our benchmark consists of programs written in Python, a general-purpose language that captures a broader range of real-world algorithms and tasks.

For the main evaluation (Section 6.5.1), we provide

Source	Functions	Lines of Code		
		Min	Max	Avg
HumanEval+	78	7	56	18.5
MBPP+	131	2	21	3.9
APPS	905	2	74	9.5
Annotated	1114	2	74	9.5
Anonymized	1114	2	74	9.5

Table 6.1: Number of functions and lines of code statistics for each benchmark source across both dataset versions.

Model	Annotated Dataset			Anonymized Dataset		
	#I/O	# Oracle	Success (%)	#I/O	# Oracle	Success (%)
Llama-3.2-3B	28.3	1.9	11.0	29.3	2.0	4.8
Mixtral-8x7B	27.4	1.9	20.3	28.5	1.9	12.0
Llama-3.1-8B	28.0	1.8	19.3	28.6	1.9	13.7
Mixtral-8x22B	26.7	1.8	25.1	28.1	1.9	15.0
QwQ-32B	24.6	1.8	20.0	25.7	1.9	15.4
Qwen2.5-7B	26.9	1.8	29.2	28.3	1.9	15.8
Llama-3.2-11B	27.3	1.8	24.9	28.3	1.9	16.1
gpt-4o-mini	27.0	1.8	26.1	27.9	1.8	18.5
Llama-3.2-90B	26.2	1.8	28.4	27.7	1.9	19.7
Llama-3.1-70B	26.9	1.8	30.1	27.9	1.9	20.0
Qwen2.5-72B	25.5	1.7	30.1	27.1	1.8	21.6
Llama-3.1-405B	24.2	1.7	38.6	26.0	1.8	26.7
gpt-4o	23.4	1.7	37.8	25.2	1.8	28.7
DeepSeek-V3	23.7	1.7	37.7	25.1	1.8	29.5
claude3.7-sonnet	23.6	1.7	39.0	24.6	1.7	33.8
DeepSeek-R1	18.6	1.6	49.8	20.3	1.7	41.3
o1-mini	21.0	1.6	53.2	21.5	1.6	47.7
o3-mini	15.6	1.5	59.5	16.0	1.6	52.7

Table 6.2: Success rates of LLMs on CodeARC using both annotated and anonymized datasets. We also report the average number of observed input-output examples and oracle invocations. All open-source models are instruction-tuned.

10 initial input-output examples and set the query budget to 30 input-output pairs and 2 oracle calls ($B_{\text{io}} = 30$, $B_{\text{oracle}} = 2$), chosen based on practical constraints such as API cost and runtime. Section 6.5.3 reports ablation studies on both budgets. For supervised fine-tuning on synthetic reasoning trajectories, we use gpt-4o as the teacher model and LLaMA-3.1-8B-Instruct as the student. We use two state-of-the-art differential testing tools, Pynguin [177] and Mokav [71]. Empirically, the two tools produce identical outcomes on 75.4% of problems. Mokav detects additional bugs in 18.8% of cases missed by Pynguin, while the reverse holds in only 5.9%. Overall, using both tools yields a more reliable approximation of correctness in the absence of a perfect oracle.

6.5 Results

6.5.1 Main Results

Table 6.2 shows the results for 18 large language models on CodeARC. The order is sorted based on the success rate on the anonymized dataset. We also report the average number of observed input-output examples and oracle invocations. Our findings are as follows:

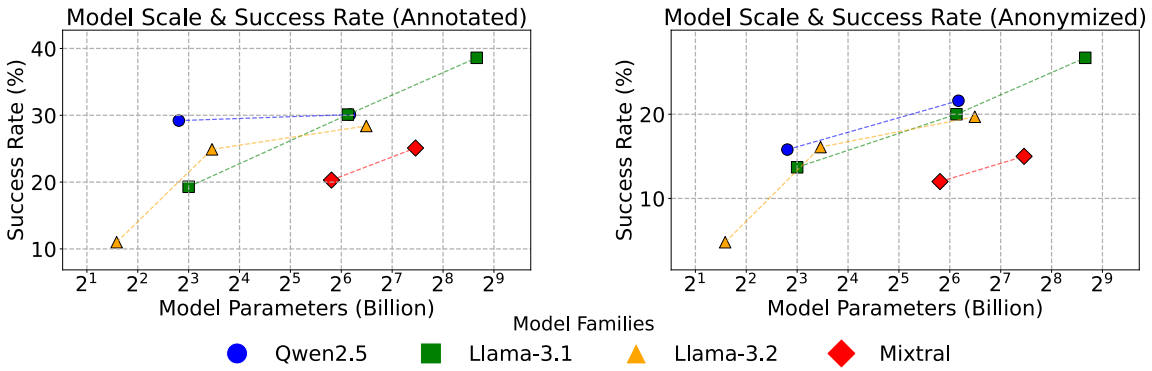


Figure 6.2: Scaling trend on CodeARC.

Reasoning models perform best. Reasoning models (o3-mini, o1-mini, DeepSeek-R1) achieve the highest success rates, all exceeding 40% on the anonymized dataset. They also require fewer I/O examples and oracle calls, indicating greater accuracy and efficiency.

CodeARC is a challenging benchmark. Among the 18 evaluated models, only OpenAI's o3-mini achieves over 50% success on both datasets (i.e., 59.5% on the annotated and 52.7% on the anonymized) while all other models fall short of this threshold. This underscores the difficulty of the task and reveals the limitations of current models in inductive reasoning.

Anonymization of function names reduces performance, but trends persist. All models show a modest drop in success rate on the anonymized dataset. However, the overall ranking remains largely consistent. This suggests that while the presence of meaningful function names provides some benefit, strong inductive reasoning remains the main factor behind high performance on this synthesis benchmark.

Scaling up model size improves performance. Larger models generally achieve better performance, as shown in Figure 6.2 (log-scale x-axis). All model families exhibit scaling trends, though with varying consistency. Llama-3.1 scales steadily, while Llama-3.2 plateaus at larger sizes, likely due to its multimodal focus. Qwen2.5 shows clearer scaling on anonymized data, where reasoning is required over memorization, highlighting model size's impact on generalization.

6.5.2 Do Initial Input-Output Examples Underspecify the Target Function?

To assess whether 10 input-output examples [155] suffice to specify the target function, we evaluate the first synthesized function of o3-mini, i.e., the strongest model in our study. Functions that pass the initial examples but fail under oracle testing indicate under-specification, motivating the need

Metric	# Problems (%)
Pass1: Initial I/O Examples	1506 (67.6%)
Pass2: Testing Oracle	866 (38.9%)
Δ = **Pass1** − **Pass2**	**640 (28.7%)**

Table 6.3: Number of problems (in both datasets) where the synthesized function passes the initial examples compared to the oracle.

Model	Success Rate (%)		
	10 I/O	20 I/O	30 I/O
o1-mini	43.7	49.6	50.5
o3-mini	51.3	53.8	56.1

Table 6.4: Success rates (%) with varying budgets on the observable input-output examples (on both datasets).

for additional examples or oracle-guided feedback. As shown in Table 6.3, 67.6% pass the initial 10 input-output example tests, but only 38.9% pass the oracle, revealing 640 cases (28.7%) where the initial examples fail to uniquely specify the target function. These findings show that initial input-output examples often under-specify program behavior, motivating additional queries and oracle-guided feedback for reliable evaluation. This motivates the design of our interactive evaluation protocol.

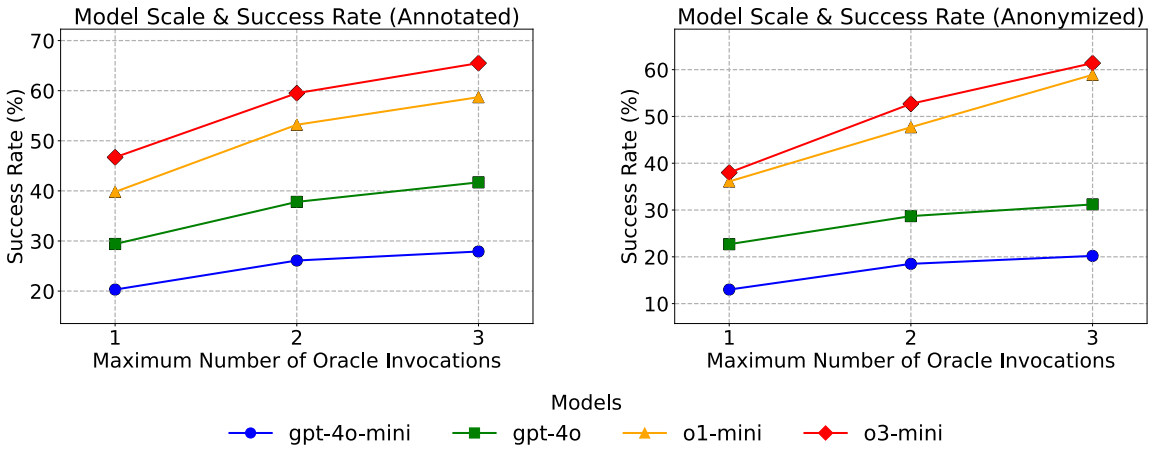


Figure 6.3: Success rates (%) of LLM models across varying numbers of oracle invocations.

6.5.3 Ablation Study: Input-Output Queries and Oracle Feedback

We perform two ablation studies to evaluate how varying the budgets for querying ground-truth functions and invoking the oracle impacts performance.

Effect of Input-Output Query Budget. We evaluate o3-mini and o1-mini with input-output budgets of 10, 20, and 30, using the same setup as Section 6.5.1. Table 6.4 shows that success rates improve consistently with more examples.

Effect of Oracle Invocation Budget. We vary the number of allowed oracle invocations and report success rates in Figure 6.3 for four models on both datasets. More oracle calls consistently improve performance, showing that counterexamples from differential testing are valuable for guiding iterative refinement.

These results demonstrate that incorporating both querying mechanisms and oracle feedback consistently enhances overall performance. This improvement underscores the importance of adopting an interactive evaluation protocol rather than relying solely on static, one-shot evaluation approaches.

6.5.4 Performance of Fine-Tuned Models

Table 6.5 shows that fine-tuning the LLaMA-3.1-8B-Instruct model on curated synthesis traces yields consistent improvements across both datasets. The larger gain on the annotated variant suggests that fine-tuning is particularly effective when semantically informative identifiers are present. Notably, this performance gap emerges despite both datasets being evaluated under the same model architecture and training methodology discussed in Section 6.3.4. These results indicate that while fine-tuning helps, there remains substantial room for further improvement. This suggests that future research may focus on enhancing the quality and diversity of the fine-tuning dataset, particularly for the anonymized variant, where gains are more limited. Another promising direction is to explore reinforcement learning approaches [269] that optimize for higher synthesis success rates, potentially overcoming limitations of supervised fine-tuning alone.

Dataset	Success Rate (%)		
	Base Model	Fine-Tuned	Rel. Δ
Annotated	19.3	25.3	+31%
Anonymized	13.7	15.0	+9.5%

Table 6.5: Success rates of LLaMA-3.1-8B-Instruct and its fine-tuned variant on annotated and anonymized datasets. Fine-tuning improves performance, especially on the annotated dataset.

To assess whether fine-tuning affects general code generation capabilities, we additionally evaluate the models on BigCodeBench [323]. Unlike our inductive program synthesis benchmark, BigCodeBench measures traditional code generation from natural language descriptions, covering diverse function calls and library usage. Our synthetic fine-tuning dataset is constructed independently, with no overlap with BigCodeBench, providing a clean test of the fine-tuned model's performance on other coding tasks.

On BigCodeBench, the base model achieves a pass@1 score of 40.1%, while the fine-tuned model attains 39.6%, a marginal decrease of 0.5 percentage points. This aligns with prior findings on catastrophic forgetting [139], a well-documented phenomenon in which training on a new task can impair performance on previously learned ones. We consider this small degradation acceptable given the gains in inductive program synthesis.

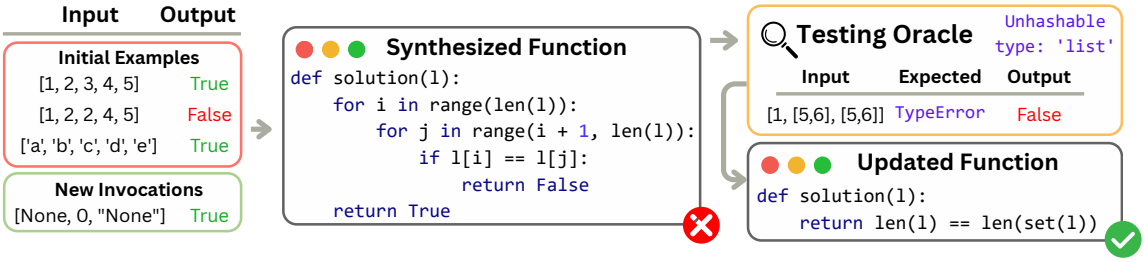


Figure 6.4: Case Study. The model queries edge cases, synthesizes a comparison function, receives a counterexample from the oracle, and corrects it with a set-based solution.

6.5.5 Case Study

Figure 6.4 shows an interaction trace from our benchmark. The model starts by querying the ground-truth function with edge-case inputs, aiming to probe its behavior beyond the initial examples. It then synthesizes a candidate solution using pairwise comparisons, which passes the given examples but fails on a counterexample with unhashable elements. From the error message, the model correctly infers that the ground-truth function raises a `TypeError`, while its own does not. On the second attempt, it reasons that using a set simplifies the uniqueness check and synthesizes the correct set-based function. This case illustrates how the model combines function invocation and oracle feedback to perform inductive program synthesis.

6.6 Discussion

6.6.1 Results on BigCodeBench for Broader Domain Coverage

Our interactive protocol is domain-agnostic and readily extensible to any Python program. To further increase the domain coverage, we extend the original CodeARC dataset with 200 Python functions randomly sampled from BigCodeBench [323], which includes problems involving scientific computing (e.g., NumPy, SciPy, pandas), machine learning libraries (e.g., sklearn), visualization (e.g., matplotlib), and system libraries. This addition enhances domain diversity beyond traditional algorithm-focused problems.

Table 6.6 shows that reasoning models (o3-mini, o1-mini) achieve lower success rates on BigCodeBench compared to their performance on the original CodeARC benchmarks. In contrast, non-reasoning models perform better on BigCodeBench. We attribute this to the nature of BigCodeBench tasks, which often involve domain-specific APIs and library usage, where non-reasoning models perform better. Notably, our fine-tuned model shows a substantial gain over the base model on BigCodeBench, indicating that the model fine-tuned on synthesis reasoning traces can generalize to unseen datasets.

Model	CodeARC (Original)	BigCodeBench
LLaMA-3.1-8B Base	19.3	32.5
Fine-tuned LLaMA	25.3	40.0
gpt-4o	37.8	46.5
o1-mini	53.2	40.0
o3-mini	59.5	43.0

Table 6.6: Success rates (%) on the original CodeARC benchmark and BigCodeBench, which covers diverse domains such as scientific computing and machine learning, extending beyond algorithm-focused problems.

6.6.2 Impact of Providing Access to a Python Interpreter

Our default evaluation restricts access to external tools such as a Python interpreter, though the CodeARC protocol can easily incorporate them via additional actions. To assess the impact, we added a code execution action, updated the prompts, and evaluated 100 sampled problems from the dataset.

Results shown in Table 6.7 suggest that providing access to a Python interpreter does not uniformly improve model performance. For instance, while o3-mini benefits modestly (+2.0%), gpt-4o's performance slightly decreases (-1.5%). This outcome highlights that expanding the agent's action space may sometimes lead to suboptimal behavior.

Model	w/o Interpreter	w/ Interpreter
gpt-4o	48.5	46.0
o3-mini	69.0	71.0

Table 6.7: Success rates (%) with and without access to a Python interpreter.

We note that internal code simulation is a natural part of the inductive program synthesis process. Our benchmark is designed to evaluate inductive reasoning capability, and we believe that requiring models to reason without relying on external execution tools remains a meaningful and challenging setting.

6.7 Conclusion

We introduce CodeARC, a new framework for evaluating LLMs on inductive program synthesis through interactive input generation and self-correction. Unlike static protocols, CodeARC allows agents to query a ground truth function and use a differential testing oracle to get feedback for iterative refinement. Designed to assess inductive reasoning from input-output examples, our benchmark covers 1114 diverse and general-purpose functions and evaluates 18 language models. The best-performing model, OpenAI o3-mini, achieves a success rate of 52.7%. Fine-tuning LLaMA-3.1-8B-Instruct on curated synthesis traces results in a 31% relative performance gain. CodeARC provides a more realistic and challenging testbed for evaluating LLM-based inductive program synthesis.

Chapter 7

Conclusions

7.1 Summary

This dissertation investigates two complementary directions at the intersection of large language models and software systems. The first direction examines how LLMs can be leveraged as optimizers to improve the performance of software systems. The second direction focuses on evaluating the limitations of LLMs in program reasoning through carefully designed benchmarks and evaluation protocols. Together, these efforts provide an integrated perspective on both the opportunities and challenges of applying language models to core problems in systems and programming languages.

In the first part of the dissertation, we explore the use of LLMs across several performance-critical system challenges. In Chapter 2, we demonstrate that LLMs can generate high-performance mapping policies for parallel programs in the Legion runtime. Chapter 3 shows that LLMs can serve as effective superoptimizers for assembly programs when fine-tuned using reinforcement learning. Chapter 4 investigates the use of LLMs for invariant synthesis to accelerate program verification.

In the second part of the dissertation, we focus on exposing the limitations of LLMs in program reasoning through benchmark design. In Chapter 5, we introduce program equivalence checking as a new benchmark for evaluating reasoning about program semantics. Finally, Chapter 6 studies the inductive reasoning capabilities of LLMs through an interactive protocol on program synthesis.

7.2 Future Research Directions

Bridging LLM Code Generation and Formal Verification. The strong generation capabilities of LLMs position formal verification to become increasingly important, establishing the "generate-then-verify" paradigm as a dominant approach. While formal verification has historically been applied to restricted domains—such as verifying access control policies in Amazon Web Services [14], proving the correctness of cryptographic implementations [69], and establishing the equivalence of

GPU kernels [66]—the advent of LLMs creates new opportunities for broader application. At the same time, LLM-generated code lacks formal correctness guarantees, making verification even more essential. For instance, the superoptimization work in this dissertation relies on test-based validation, which, despite achieving high coverage, cannot provide the strong assurances afforded by formal verification. An important direction for future work is the development of frameworks that combine the generative capabilities of LLMs with the rigor of formal verification. Such frameworks would allow LLMs to propose candidate optimizations while formal methods ensure their soundness and correctness. This integration could enable scalable and reliable program optimization where the creativity of neural models is constrained by the guarantees of formal proof systems. We envision the development of specialized verifiers tailored to diverse code generation domains: assembly programs, GPU kernels at multiple abstraction levels (DSL, PTX, SASS), and performance-critical systems including OS kernels [311], database management systems, and network stacks. By pairing LLM-generated optimizations with domain-specific verification tools, we can achieve both the flexibility of learned approaches and the soundness guarantees required for production deployment.

LLM-Driven System Design and Self-Evolution. A long-term and compelling vision is the development of intelligent software systems that can self-improve and self-evolve. The automated mapper generation work presented in this dissertation represents early steps in that direction, demonstrating that systems can be optimized with minimal human intervention when LLMs are paired with appropriate abstractions and feedback mechanisms. However, significant challenges and opportunities for further automation remain. For instance, can LLMs propose suitable abstractions and implement compilers on top of them, rather than relying on human-designed languages as in the mapper generation chapter? Can LLMs learn to ask clarification questions when given vague natural language descriptions of problems? Can they act as system designers, proactively identifying ambiguities and proposing design alternatives? How should we design human-in-the-loop systems that allow experts to provide guidance at critical decision points? Can LLMs create their own feedback loops and iteratively refine their designs based on observed outcomes? There is substantial ongoing research exploring whether LLMs can meaningfully contribute as mathematicians or AI researchers [80]. We are interested in a parallel question: can LLM agents become systems researchers—designing and implementing systems, self-evolving iteratively over time, and maintaining software in response to new feature requests? And if so, how can we design such agents while ensuring that human experts retain meaningful control and can engage in effective AI-guided collaboration?

Programming Language Design for Safer Code Generation. As LLMs become increasingly capable of generating software, we anticipate a future in which language models write a substantial portion of production code. This shift introduces new risks: LLMs can introduce subtle bugs, security vulnerabilities, and undefined behaviors that may go undetected without careful review. To mitigate these risks, it becomes essential to design programming languages and abstractions that constrain the

space of errors a model can make, reducing the likelihood of unsafe code by construction. Current LLMs have been trained predominantly on human-written code in dynamically typed or weakly typed languages such as Python and JavaScript. However, we observe a growing trend toward languages with stronger static guarantees, such as Rust, which enforces memory safety through its ownership system, despite their steeper learning curves. We envision that future programming languages will push this trajectory further, incorporating even more expressive type systems, dependent types, and built-in verification capabilities that allow compilers to reject entire classes of errors before code ever executes. Such languages would enable LLMs to produce code with stronger correctness and safety properties by design, shifting the burden of verification from runtime testing to compile-time guarantees. A practical challenge is that these safer languages may initially lack the large-scale training corpora available for mainstream languages. However, techniques such as self-play [62], synthetic data generation, and curriculum learning could enable LLMs to acquire proficiency in these languages over time.

Semantic Learning of Programming Languages by LLMs. Language models are trained on large corpora of code, yet as demonstrated in our chapter on equivalence checking, they do not fully understand the semantics of programming languages. This observation raises two compelling research directions. The first concerns evaluation: can we assess whether LLMs truly understand a language by testing their ability to infer the language specification from example programs? Specifically, given a sufficient number of programs, can an LLM learn the syntax, type system, and operational semantics of a language, produce a formal specification, and generate a correct interpreter? Success on such tasks would provide strong evidence of deep semantic understanding beyond superficial pattern matching. The second direction concerns training methodology: what is the most effective way to teach LLMs a new programming language? There remains limited understanding of which types of training data are most effective for developing code capabilities, and how best to filter, curate, and structure such data. We believe future research can address this by designing small, novel languages and systematically experimenting with different training strategies (e.g., varying data composition, curriculum design, and learning objectives) to identify the most efficient approaches for language acquisition. Additionally, understanding how to optimally allocate computational resources across pre-training, mid-training, and post-training phases remains an open and important question for advancing LLM capabilities in learning programming languages.

Bibliography

[1] Hyperfine, 2025.

[2] The next horizon of system intelligence, 2025.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[4] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[5] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[6] Ali AhmadiTeshnizi, Wenzhi Gao, Herman Brunborg, Shayan Talaei, and Madeleine Udell. OptiMUS-0.3: Using large language models to model and solve optimization problems at scale. *Submitted*, 2024.

[7] Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling using MIP solvers and large language models. In *International Conference on Machine Learning (ICML)*, 2024.

[8] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271. IEEE, 2023.

[9] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[10] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[11] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *16th International Conference on Parallel and Distributed Systems*, pages 291–298, Shangai, China, December 2010. IEEE.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *European Conference on Parallel Processing*, pages 863–874. Springer, 2009.

[13] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[14] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for aws access policies using smt. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–9. IEEE, 2018.

[15] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 13–24, 2020.

[16] Sahar Badihi, Yi Li, and Julia Rubin. Eqbench: A dataset of equivalent and non-equivalent program pairs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 610–614. IEEE, 2021.

[17] Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Crane: Reasoning with constrained llm generation, 2025.

[18] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, et al. Pathways: Asynchronous distributed dataflow for ml. *Proceedings of Machine Learning and Systems*, 4:430–449, 2022.

[19] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems*, 37:15612–15645, 2024.

[20] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-turn rl for generating cuda kernels. *arXiv preprint arXiv:2507.11948*, 2025.

[21] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[22] Dirk Beyer and Jan Strejček. Improvements in software verification and witness validation: Sv-comp 2025. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–186. Springer, 2025.

[23] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit Seshia, and Alvin Cheung. Verified code transpilation with llms. *Advances in Neural Information Processing Systems*, 37:41394–41424, 2024.

[24] Samuel Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 632–642, 2015.

[25] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

[27] Rudy Bunel, Alban Desmaison, M Pawan Kumar, Philip HS Torr, and Pushmeet Kohli. Learning to superoptimize programs. *arXiv preprint arXiv:1611.01787*, 2016.

[28] Chengkun Cai, Xu Zhao, Haoliang Liu, Zhongyu Jiang, Tianfang Zhang, Zongkai Wu, Jenq-Neng Hwang, and Lei Li. The role of deductive and inductive reasoning in large language models. *arXiv preprint arXiv:2410.02892*, 2024.

[29] Lynn Elliot Cannon. *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969.

[30] Thomas J. Watson IBM Research Center, F.E. Allen, and J. Cocke. *A Catalogue of Optimizing Transformations*. IBM Thomas J. Watson Research Center, 1971.

[31] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342*, 2023.

[32] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[33] Liangyu Chen, Bo Li, Sheng Shen, Jingkang Yang, Chunyuan Li, Kurt Keutzer, Trevor Darrell, and Ziwei Liu. Large language models are visual reasoning coordinators. *Advances in Neural Information Processing Systems*, 36, 2024.

[34] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[35] Minyu Chen, Guoqiang Li, Ling-I Wu, and Ruibang Liu. Dce-llm: Dead code elimination with large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 9942–9955, 2025.

[36] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[37] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[38] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[39] Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next autodiff: Generative optimization with rich feedback, execution traces, and llms. *arXiv preprint arXiv:2406.16218*, 2024.

[40] Jaeyoung Choi, David W Walker, and Jack J Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[41] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[42] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040, 2019.

[43] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. Sound loop superoptimization for google native client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 313–326, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[45] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[46] Michael Colon, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. *Computer-aided Verification: Proceedings*, page 420, 2003.

[47] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[48] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, 1979.

[49] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[50] Chris Cummins, Volker Seeker, Dejan Grubisic, Mostafa Elhoushi, Youwei Liang, Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Kim Hazelwood, Gabriel Synnaeve, et al. Large language models for compiler optimization. *arXiv preprint arXiv:2309.07062*, 2023.

[51] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. *arXiv preprint arXiv:2407.02524*, 2024.

[52] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Llm compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, pages 141–153, 2025.

[53] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[54] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*, pages 127–147. Springer, 2017.

[55] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[56] Haowei Deng, Runzhou Tao, Yuxiang Peng, and Xiaodi Wu. A case for synthesis of recursive quantum unitary programs. *Proceedings of the ACM on Programming Languages*, 8(POPL):1759–1788, 2024.

[57] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[58] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.

[59] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024.

[60] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. Proving query equivalence using linear integer arithmetic. *Proceedings of the ACM on Management of Data*, 1(4):1–26, 2023.

[61] Yangruibo Ding, Jinjun Peng, Marcus J Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. *arXiv preprint arXiv:2406.01006*, 2024.

[62] Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

[63] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

[64] Mingzhe Du, Anh Tuan Luu, Bin Ji, Qian Liu, and See-Kiong Ng. Mercury: A code efficiency benchmark for code large language models. *arXiv preprint arXiv:2402.07844*, 2024.

[65] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

[66] Kshitij Dubey, Benjamin Driscoll, Anjiang Wei, Neeraj Kayal, Rahul Sharma, and Alex Aiken. Equivalence checking of ml gpu kernels. *arXiv preprint arXiv:2511.12638*, 2025.

[67] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. Ilinva: Using abduction to generate loop invariants. In *Frontiers of Combining Systems: 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings 12*, pages 77–93. Springer, 2019.

[68] Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.

[69] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Operating Systems Review*, 54(1):23–30, 2020.

[70] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[71] Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. Mokav: Execution-driven differential testing with llms. *arXiv preprint arXiv:2406.10375*, 2024.

[72] DOE Explains Exascale Computing, 2025. https://www.energy.gov/science/doe-explainsexascale-computing.

[73] P Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P Madhusudan. Horn-ice learning for synthesizing invariants and contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[74] Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. Dryvr: Data-driven verification and compositional reasoning for automotive systems. In *International Conference on Computer Aided Verification*, pages 441–461. Springer, 2017.

[75] Yixiong Fang, Tianran Sun, Yuling Shi, Min Wang, and Xiaodong Gu. Lastingbench: Defend benchmarks against knowledge leakage. *arXiv preprint arXiv:2506.21614*, 2025.

[76] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, volume 0 of *SC '06*, page 83–es, New York, NY, USA, 2006. Association for Computing Machinery.

[77] Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*, pages 251–269. Springer, 2018.

[78] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 349–360, 2014.

[79] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 804–817, 2023.

[80] Tony Feng, Trieu H Trinh, Garrett Bingham, Dawsen Hwang, Yuri Chervonyi, Junehyuk Jung, Joonkyung Lee, Carlo Pagano, Sang-hyun Kim, Federico Pasqualotto, et al. Towards autonomous mathematics research. *arXiv preprint arXiv:2602.10177*, 2026.

[81] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

[82] Charles R Ferenbaugh. Pennant: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience*, 27(17):4555–4572, 2015.

[83] Jack Feser, Işıl Dillig, and Armando Solar-Lezama. Inductive program synthesis guided by observational program similarity. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):912–940, 2023.

[84] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

[85] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, 2002.

[86] Daniel Flook. Python variable renaming tool, 2025. https://github.com/dflook/python-minifier.

[87] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

[88] Mikhail R Gadelha, Felipe R Monteiro, Jeremy Morse, Lucas C Cordeiro, Bernd Fischer, and Denis A Nicole. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 888–891, 2018.

[89] Juan J Galvez, Nikhil Jain, and Laxmikant V Kale. Automatic topology mapping of diverse large-scale parallel applications. In *Proceedings of the International Conference on Supercomputing*, pages 1–10, 2017.

[90] Zeyu Gao, Hao Wang, Yuanda Wang, and Chao Zhang. Virtual compiler is all you need for assembly code search. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3040–3051, 2024.

[91] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 69–87. Springer, 2014.

[92] Pranav Garg, Daniel Neider, Parthasarathy Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 51(1):499–512, 2016.

[93] Dejan Grubisic, Chris Cummins, Volker Seeker, and Hugh Leather. Compiler generated feedback for large language models. *arXiv preprint arXiv:2403.14714*, 2024.

[94] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

[95] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[96] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation: 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings 10*, pages 120–135. Springer, 2009.

[97] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[98] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[99] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

[100] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 262–276. Springer, 2009.

[101] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 365–382. Springer, 2018.

[102] Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*, 2023.

[103] Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. Absynthe: Abstract interpretation-guided synthesis. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1584–1607, 2023.

[104] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[105] Ameer Haj-Ali, Qijing Jenny Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. *Proceedings of Machine Learning and Systems*, 2:70–81, 2020.

[106] Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, et al. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

[107] Zain Hasan. How to evaluate and benchmark Large Language Models (LLMs), 2025. https://www.together.ai/blog/evaluate-and-benchmark-llms.

[108] Brett K Hayes, Evan Heit, and Haruka Swendsen. Inductive reasoning. *Wiley interdisciplinary reviews: Cognitive science*, 1(2):278–292, 2010.

[109] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. Hpx–an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo*, 5, 2017.

[110] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[111] Namgyu Ho, Laura Schmid, and Se-Young Yun. Large language models are reasoning teachers. *arXiv preprint arXiv:2212.10071*, 2022.

[112] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[113] Hossein Hojjat and Philipp Rümmer. The eldarica horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–7. IEEE, 2018.

[114] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[115] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[116] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie Zhang. Effilearner: Enhancing efficiency of generated code via self-optimization. *Advances in Neural Information Processing Systems*, 37:84482–84522, 2024.

[117] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

[118] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

[119] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024.

[120] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320, 2021.

[121] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[122] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. Repository-level compositional code translation and validation. *arXiv preprint arXiv:2410.24117*, 2024.

[123] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[124] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024.

[125] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 402–416, 2022.

[126] Ranjit Jhala and Kenneth L McMillan. A practical and complete approach to predicate refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 459–473. Springer, 2006.

[127] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92, 2009.

[128] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[129] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[130] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

[131] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[132] Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Leveraging llms for program verification. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pages 107–118. IEEE, 2024.

[133] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.

[134] Shyam Sundar Kannan, Vishnunandan LN Venkatesh, and Byung-Cheol Min. Smart-llm: Smart multi-agent robot task planning using large language models. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12140–12147. IEEE, 2024.

[135] Michael Karr. Affine relationships among variables of a program. *Acta informatica*, 6(2):133–151, 1976.

[136] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[137] Jason R Koenig, Oded Padon, and Alex Aiken. Adaptive restarts for stochastic synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 696–709, 2021.

[138] Jason R. Koenig, Oded Padon, and Alex Aiken. Replication package for article: Adaptive restarts for stochastic synthesis, 2021.

[139] Suhas Kotha, Jacob Mitchell Springer, and Aditi Raghunathan. Understanding catastrophic forgetting in language models via implicit inference. In *The Twelfth International Conference on Learning Representations*, 2024.

[140] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–22, 2019.

[141] Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074*, 2024.

[142] Shuvendu K Lahiri and Randal E Bryant. Predicate abstraction with indexed predicates. *ACM Transactions on Computational Logic (TOCL)*, 9(1):4–es, 2007.

[143] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[144] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020.

[145] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[146] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[147] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. Sling: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 788–801, 2019.

[148] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008, 2023.

[149] Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. *Advances in Neural Information Processing Systems*, 37:57619–57641, 2024.

[150] Jiachun Li, Pengfei Cao, Zhuoran Jin, Yubo Chen, Kang Liu, and Jun Zhao. Mirage: Evaluating and explaining inductive reasoning process in language models. *arXiv preprint arXiv:2410.09542*, 2024.

[151] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. Automatic loop-invariant generation and refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 782–792. IEEE, 2017.

[152] Junkai Li, Siyu Wang, Meng Zhang, Weitao Li, Yunghwei Lai, Xinhui Kang, Weizhi Ma, and Yang Liu. Agent hospital: A simulacrum of hospital with evolvable medical agents. *arXiv preprint arXiv:2405.02957*, 2024.

[153] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[154] Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

[155] Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2025.

[156] Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025.

[157] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[158] Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh James Leather, et al. Learning compiler pass orders using coreset and normalized value prediction. In *International Conference on Machine Learning*, pages 20746–20762. PMLR, 2023.

[159] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, pages 1–1, 2010.

[160] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[161] Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. Towards general loop invariant generation: a benchmark of programs with memory manipulation. *Advances in Neural Information Processing Systems*, 37:129120–129145, 2024.

[162] Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. Codemind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*, 2024.

[163] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*, 2023.

[164] Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. *arXiv preprint arXiv:2007.08124*, 2020.

[165] Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023.

[166] Jiawei Liu, Thanh Nguyen, Mingyue Shang, Hantian Ding, Xiaopeng Li, Yu Yu, Varun Kumar, and Zijian Wang. Learning code preference via synthetic evolution. *arXiv preprint arXiv:2410.03837*, 2024.

[167] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

[168] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[169] Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*, 2024.

[170] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[171] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

[172] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[173] Hatem Ltaief, Rabab Alomairy, Qinglei Cao, Jie Ren, Lotfi Slim, Thorsten Kurth, Benedikt Dorschner, Salim Bougouffa, Rached Abdelkhalak, and David E Keyes. Toward capturing genetic epistasis from multivariate genome-wide association studies using mixed-precision kernel ridge regression. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2024.

[174] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[175] Tyler Lu, Dávid Pál, and Martin Pál. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pages 485–492. JMLR Workshop and Conference Proceedings, 2010.

[176] Matt Luckcuck, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.

[177] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[178] Kaijing Ma, Xinrun Du, Yunran Wang, Haoran Zhang, Zhoufutu Wen, Xingwei Qu, Jian Yang, Jiaheng Liu, Minghao Liu, Xiang Yue, et al. Kor-bench: Benchmarking language models on knowledge-orthogonal reasoning tasks. *arXiv preprint arXiv:2410.06526*, 2024.

[179] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[180] Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[181] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.

[182] Nickil Maveli, Antonio Vergari, and Shay B Cohen. What can large language models capture about code functional equivalence? *arXiv preprint arXiv:2408.11081*, 2024.

[183] Kenneth L McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pages 104–118. Springer, 2010.

[184] Stephen Mell, Steve Zdancewic, and Osbert Bastani. Optimal program synthesis via abstract interpretation. *Proceedings of the ACM on Programming Languages*, 8(POPL):457–481, 2024.

[185] Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. A diverse corpus for evaluating and developing english math word problem solvers. *arXiv preprint arXiv:2106.15772*, 2021.

[186] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[187] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *International conference on machine learning*, pages 2430–2439. PMLR, 2017.

[188] Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*, 2024.

[189] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. Client-specific equivalence checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 441–451, 2018.

[190] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[191] Brad A Myers. Visual programming, programming by example, and program visualization: a taxonomy. *ACM sigchi bulletin*, 17(4):59–66, 1986.

[192] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024.

[193] Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. L2ceval: Evaluating language-to-code generation capabilities of large language models. *Transactions of the Association for Computational Linguistics*, 12:1311–1329, 2024.

[194] Allen Nie, Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. The importance of directional feedback for llm-based optimizers. *arXiv preprint arXiv:2405.16434*, 2024.

[195] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[196] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, page 1–10, USA, 2013. IEEE Computer Society.

[197] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

[198] Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. Semantic code refactoring for abstract data types. *Proceedings of the ACM on Programming Languages*, 8(POPL):816–847, 2024.

[199] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[200] Mihir Parmar, Nisarg Patel, Neeraj Varshney, Mutsumi Nakamura, Man Luo, Santosh Mashetty, Arindam Mitra, and Chitta Baral. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. *arXiv preprint arXiv:2404.15522*, 2024.

[201] Bhrij Patel, Souradip Chakraborty, Wesley A Suttle, Mengdi Wang, Amrit Singh Bedi, and Dinesh Manocha. Aime: Ai system optimization via multiple llm evaluators. *arXiv preprint arXiv:2410.03131*, 2024.

[202] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2025.

[203] Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. Refiner: Reasoning feedback on intermediate representations. *arXiv preprint arXiv:2304.01904*, 2023.

[204] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR, 23–29 Jul 2023.

[205] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

[206] Gabriel Poesia, Breno Guimarães, Fabrício Ferracioli, and Fernando Magno Quintão Pereira. Static placement of computation on heterogeneous devices. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), October 2017.

[207] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[208] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[209] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*, 2016.

[210] Elkin Arturo Betancourt Ramirez and Juan Antonio Fuentes Esparrell. Artificial intelligence (ai) in education: Unlocking the perfect synergy for learning. *Educational Process: International Journal*, 13(1):35–51, 2024.

[211] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 280–291, New York, NY, USA, 2008. Association for Computing Machinery.

[212] Daniel Riley and Grigory Fedyukovich. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 607–619, 2022.

[213] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[214] Sebastian Ruder. Challenges and Opportunities in NLP Benchmarking, 2021. http://ruder.io/nlp-benchmarking.

[215] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

[216] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[217] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

[218] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[219] Frank Schüssele, Manuel Bentele, Daniel Dietsch, Matthias Heizmann, Xinyu Jiang, Dominik Klumpp, and Andreas Podelski. Ultimate automizer and the abstraction of bitwise operations: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 418–423. Springer, 2024.

[220] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[221] Thiago SFX Teixeira, Alexandra Henzinger, Rohan Yadav, and Alex Aiken. Automated mapping of task-based programs onto distributed and heterogeneous machines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[222] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[223] Yijia Shao, Yucheng Jiang, Theodore A Kanell, Peter Xu, Omar Khattab, and Monica S Lam. Assisting in writing wikipedia-like articles from scratch with large language models. *arXiv preprint arXiv:2402.14207*, 2024.

[224] Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, et al. Case2code: Learning inductive reasoning with synthetic data. *arXiv e-prints*, pages arXiv–2407, 2024.

[225] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[226] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V Nori. Verification as learning geometric concepts. In *International Static Analysis Symposium*, pages 388–411. Springer, 2013.

[227] Rahul Sharma, Aditya V Nori, and Alex Aiken. Interpolants as classifiers. In *International Conference on Computer Aided Verification*, pages 71–87. Springer, 2012.

[228] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 391–406, 2013.

[229] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Conditionally correct superoptimization. *ACM SIGPLAN Notices*, 50(10):147–162, 2015.

[230] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv:2409.19256*, 2024.

[231] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[232] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

[233] Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024.

[234] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31, 2018.

[235] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279. IEEE, 2017.

[236] Rajat Singh and Srikanta Bedathur. Exploring the use of llms for sql equivalence checking. *arXiv preprint arXiv:2412.05561*, 2024.

[237] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–356, 2016.

[238] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[239] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[240] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *Euro-Par 2011 Parallel Processing: 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29-September 2, 2011, Proceedings, Part II 17*, pages 90–109. Springer, 2011.

[241] Ryan Stocks, Jorge L Galvez Vallejo, CY Fiona, Calum Snowdon, Elise Palethorpe, Jakub Kurzak, Dmytro Bykov, and Giuseppe MJ Barca. Breaking the million-electron and 1 eflop/s barriers: Biomolecular-scale ab initio molecular dynamics using mp2 potentials. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2024.

[242] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code retrieval and reranking, 2025.

[243] Tarun Suresh, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Is the watermarking of llm-generated code robust?, 2025.

[244] Tarun Suresh, Nalin Wadhwa, Debangshu Banerjee, and Gagandeep Singh. Beaver: An efficient deterministic llm verifier. *arXiv preprint arXiv:2512.05439*, 2025.

[245] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[246] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. *arXiv preprint arXiv:1811.00937*, 2018.

[247] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K Lahiri. Llm-vectorizer: Llm-based verified loop vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 137–149, 2025.

[248] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[249] Ali TehraniJamsaz, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *arXiv preprint arXiv:2410.20527*, 2024.

[250] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 697–709, 2022.

[251] Zhao Tian, Honglin Shu, Dong Wang, Xuejie Cao, Yasutaka Kamei, and Junjie Chen. Large language models for equivalent mutant detection: How far are we? In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1733–1745, 2024.

[252] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[253] Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. Itergen: Iterative semantic-aware structured llm generation with backtracking, 2025.

[254] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation, 2024.

[255] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[256] Robert A Van De Geijn and Jerrell Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[257] Rob F Van der Wijngaart and Timothy G Mattson. The parallel research kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[258] Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. *Formal Methods in System Design*, 63(1):81–109, 2024.

[259] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466, 2017.

[260] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. Visualization by example, 2019.

[261] Xiao Wang, Siyan Liu, Aristeidis Tsaris, Jong-Youl Choi, Ashwin M Aji, Ming Fan, Wei Zhang, Junqi Yin, Moetasim Ashfaq, Dan Lu, et al. Orbit: Oak ridge base foundation model for earth

system predictability. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2024.

[262] Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 75–84, New York, NY, USA, 2009. Association for Computing Machinery.

[263] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

[264] Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, et al. Equibench: Benchmarking large language models' reasoning about program semantics via equivalence checking. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33856–33869, 2025.

[265] Anjiang Wei, Allen Nie, Thiago S. F. X. Teixeira, Rohan Yadav, Wonchan Lee, Ke Wang, and Alex Aiken. Improving parallel program performance with LLM optimizers via agent-system interfaces. In *Forty-second International Conference on Machine Learning*, 2025.

[266] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A multi-agent system for gpu kernel performance optimization. *arXiv preprint arXiv:2509.07506*, 2025.

[267] Anjiang Wei, Tarun Suresh, Jiannan Cao, Naveen Kannan, Yuheng Wu, Kai Yan, Thiago S. F. X. Teixeira, Ke Wang, and Alex Aiken. CodeARC: Benchmarking reasoning capabilities of LLM agents for inductive program synthesis. In *Second Conference on Language Modeling*, 2025.

[268] Anjiang Wei, Tarun Suresh, Tianran Sun, Haoze Wu, Ke Wang, and Alex Aiken. Quokka: Accelerating program verification with llms via invariant synthesis, 2026.

[269] Anjiang Wei, Tarun Suresh, Huanmi Tan, Yinglun Xu, Gagandeep Singh, Ke Wang, and Alex Aiken. Improving assembly code performance with large language models via reinforcement learning, 2025.

[270] Anjiang Wei, Huanmi Tan, Tarun Suresh, Daniel Mendoza, Thiago SFX Teixeira, Ke Wang, Caroline Trippel, and Alex Aiken. Vericoder: Enhancing llm-based rtl code generation through functional correctness validation. *arXiv preprint arXiv:2504.15659*, 2025.

[271] Anjiang Wei, Yuheng Wu, Yingjia Wan, Tarun Suresh, Huanmi Tan, Zhanke Zhou, Sanmi Koyejo, Ke Wang, and Alex Aiken. SATBench: Benchmarking LLMs' logical reasoning via automated puzzle generation from SAT formulas. In Christos Christodoulopoulos, Tanmoy

Chakraborty, Carolyn Rose, and Violet Peng, editors, *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33820–33837, Suzhou, China, November 2025. Association for Computational Linguistics.

[272] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[273] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

[274] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.

[275] Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on parallel and distributed systems*, 2(4):452–471, 1991.

[276] Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International conference on machine learning*, pages 11193–11204. PMLR, 2021.

[277] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 406–417, 2024.

[278] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations*, 2024.

[279] Jie JW Wu, Manav Chaudhary, Davit Abrahamyan, Arhaan Khaku, Anjiang Wei, and Fatemeh H Fard. Clarifycoder: Clarification-aware fine-tuning for programmatic problem solving. *arXiv preprint arXiv:2504.16331*, 2025.

[280] Mengdi Wu, Xinhao Cheng, Shengyu Liu, Chunan Shi, Jianan Ji, Kit Ao, Praveen Velliengiri, Xupeng Miao, Oded Padon, and Zhihao Jia. Mirage: A multi-level superoptimizer for tensor programs. *arXiv preprint arXiv:2405.05751*, 2024.

[281] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[282] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[283] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

[284] Chunqiu Steven Xia, Yinlin Deng, and Lingming Zhang. Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm. *arXiv preprint arXiv:2403.19114*, 2024.

[285] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. *CoRR*, 2023.

[286] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494. IEEE, 2023.

[287] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.

[288] Yijia Xiao, Edward Sun, Tianyu Liu, and Wei Wang. Logicvista: Multimodal llm logical reasoning benchmark in visual contexts. *arXiv preprint arXiv:2407.04973*, 2024.

[289] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36:41618–41650, 2023.

[290] Rongchen Xu, Fei He, and Bow-Yaw Wang. Interval counterexamples for loop invariant learning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 111–122, 2020.

[291] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Distal: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 286–300, 2022.

[292] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. *ACM SIGPLAN Notices*, 51(6):508–521, 2016.

[293] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[294] Kai Yan, Zhan Ling, Kang Liu, Yifan Yang, Ting-Han Fan, Lingfeng Shen, Zhengyin Du, and Jiecao Chen. Mir-bench: Benchmarking llm's long-context intelligence via many-shot in-context inductive reasoning. *arXiv preprint arXiv:2502.09933*, 2025.

[295] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[296] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[297] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

[298] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

[299] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. Whitefox: White-box compiler fuzzing empowered by large language models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):709–735, 2024.

[300] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563*, 2023.

[301] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[302] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[303] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1585–1608, 2024.

[304] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 106–120, New York, NY, USA, 2020. Association for Computing Machinery.

[305] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[306] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*, 2022.

[307] Shiwen Yu, Ting Wang, and Ji Wang. Loop invariant inference through smt solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 175–187, 2023.

[308] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[309] Dylan Zhang, Curt Tigges, Zory Zhang, Stella Biderman, Maxim Raginsky, and Talia Ringer. Transformer-based models are not yet perfect at learning to emulate structural recursion. *arXiv preprint arXiv:2401.12947*, 2024.

[310] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36:54769–54784, 2023.

[311] Yiying Zhang and Yutong Huang. " learned" operating systems. *ACM SIGOPS Operating Systems Review*, 53(1):40–45, 2019.

[312] Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. Llm-sql-solver: Can llms determine sql equivalence? *arXiv preprint arXiv:2312.10321*, 2023.

[313] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. Felix: Optimizing tensor programs with gradient descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 367–381, 2024.

[314] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance}

tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[315] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[316] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[317] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[318] Size Zheng, Jin Fang, Xuegui Zheng, Qi Hou, Wenlei Bao, Ningxin Zheng, Ziheng Jiang, Dongyang Wang, Jianxi Ye, Haibin Lin, et al. Tilelink: Generating efficient compute-communication overlapping kernels using tile-centric primitives. *arXiv preprint arXiv:2503.20313*, 2025.

[319] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[320] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024.

[321] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

[322] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024.

[323] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari,

Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.