Towards LLM-based optimization compilers. Can LLMs learn how to apply a single peephole optimization? Reasoning is all LLMs need!

1st Xiangxin Fang
Queen Mary University of London
London, United Kingdom
jp2019213661@qmul.ac.uk

2nd Lev Mukhanov
Queen Mary University of London
London, United Kingdom
l.mukhanov@qmul.ac.uk

arXiv:2412.12163v1 [cs.LG] 11 Dec 2024

***Abstract*—Large Language Models (LLMs) have demonstrated great potential in various language processing tasks, and recent studies have explored their application in compiler optimizations. However, all these studies focus on the conventional open-source LLMs, such as Llama2, which lack enhanced reasoning mechanisms. In this study, we investigate the errors produced by the fine-tuned 7B-parameter Llama2 model as it attempts to learn and apply a simple peephole optimization for the AArch64 assembly code. We provide an analysis of the errors produced by the LLM and compare it with state-of-the-art OpenAI models which implement advanced reasoning logic, including GPT-4o and GPT-o1 (preview). We demonstrate that OpenAI GPT-o1, despite not being fine-tuned, outperforms the fine-tuned Llama2 and GPT-4o.**

Our findings indicate that this advantage is largely due to the chain-of-thought reasoning implemented in GPT-o1. We hope our work will inspire further research on using LLMs with enhanced reasoning mechanisms and chain-of-thought for code generation and optimization.

***Index Terms*—Compiler Optimization, Large Language Model, Reasoning**

I. INTRODUCTION

Large Language Models (LLMs) have made significant advancements in various software engineering tasks, showcasing their ability to understand and generate human-like code. Numerous studies have demonstrated the capability of these models to handle various high-level programming languages effectively. In code generation, tools like GitHub Copilot [1] and OpenAI Codex [2] have proven their utility by assisting developers in creating relevant code snippets. In code translation, systems such as Facebook's TransCoder [3] and Microsoft's CodeBERT [4] have shown effectiveness in translating code between different programming languages.

The use of LLMs for compiler optimizations and code generation is a promising alternative to the classical compilers [5]. LLMs can save significant effort and cost in developing new compilers, extending the existing native and binary compilers for new ISAs, supporting new programming languages, improving decompilers [6], [7] or even deriving new ISA-specific compiler optimizations. Recent studies have demonstrated that LLMs can be applied for decompilation [6], [7] and compiling the native C code into X86 [5]. Meta went even further by releasing foundation LLMs for code compilation and optimization [8]. However, these studies focus on investigating the capabilities of conventional LLMs, such as Llama2, which lack of enhanced reasoning mechanisms, to optimize and generate the code. Moreover, these studies often tend to train on IRs or binaries where various combinations of compiler optimizations are applied. This prevents detailed analysis of the LLM errors, making it unclear to what extent these models can truly understand and optimize the code.

The main goal of our study is to investigate if advanced reasoning mechanisms, such as chain-of-thought, can help LLMs to improve their code generation and optimization capabilities. We want to understand when and what types of errors are produced by conventional LLMs and if reasoning can help to reduce these errors.

In this study, we focus specifically on peephole optimization since LLMs are sensitive to the context window size, which hinders the optimization of code samples containing many instructions [9]. We exploit the fact that peephole examines and applies optimizations to each basic block where the number of instructions is limited. Thus, we can investigate the strengths of LLMs in handling short instruction sequences, mitigating the limitations of LLMs in handling the context with a big size. We also specifically target peephole optimization since it is a fundamental part of any compiler, performing basic algebraic transformations. Thus, if LLMs cannot perform this optimization correctly, it is unlikely they will be able to handle more sophisticated optimizations.

The contribution of this study can be summarized as follows:

- we present the results of our experimental study on investigating the capabilities of the 7B-parameter Llama2 model [10] to learn and apply a basic peephole optimization. Specifically, we fine-tune the model using 100,000 AArch64 basic block samples generated by the LLVM compiler. We investigate the syntactic and output correctness of the code samples optimized by the model using 24,000 basic blocks extracted from Code-Force, Code-Jam and BigCode datasets.

- We provide a detailed study and analysis of the errors produced by the Llama2 model when applying peephole optimization. We demonstrate that the induced errors are primarily due to a lack of reasoning. We make several important observations, such as the fact that fine-tuning tends to adjust the model to fully mimic the compiler's behavior which negatively affects the model's generalization capabilities.
- We compare the accuracy of the fine-tuned Llama2 model with state-of-the-art OpenAI models, such as GPT-4o and GPT-o1 (preview). We demonstrate that GPT-o1 significantly outperforms the fine-tuned Llama2 and GPT-4o by enabling chain-of-thought.
  Finally, we demonstrate that the number of steps in chain-of-thought and inference time have a strong impact on the performance of GPT-o1.

The paper is organized as follows: Section II presents background and our framework design; Section III presents the evaluation results for our baseline model, i.e. the fine-tuned Llama2 model; Section IV presents the evaluation results for the models with enhanced reasoning; Section V discusses the limitations of our study and future research; Section VI presents related work and Section VII discusses the conclusion.

II. BACKGROUND AND FRAMEWORK DESIGN

Compiler and ISA: Peephole is an optimization technique applied at the basic block level that involves replacing the instructions with a logically equivalent set that has better performance [11]. Peephole employs numerous pattern matching algorithms to apply a wide range of optimization scenarios. Its replacements include but are not limited to the following [1] [12] :

- Null sequences – Remove instructions which results are not used.
- Combine operations – Replace multiple operations with a single equivalent one.
- Algebraic optimizations – Apply algebraic laws to simplify or rearrange instructions.
- Address mode operations – Memory address algebraic optimizations.
- Special case instructions – Use of special instructions and operands, such as zero operand.

In our study, we generate AArch64 v8 code samples using the LLVM compiler [13] (version 14.0.0) to train LLMs.

LLVM contains two default peephole implementations: *Instcombine* [14] and *Aggressive-instcombine* [15]. The first implementation includes a large number of optimizations listed above, and all the transformations are applied within an optimized basic block [14] (algorithmic complexity is *O(1)*). Meanwhile, the second implementation is more aggressive, i.e. algorithmic complexity is higher than *O(1)*, and it can also modify CFG.

Given the mentioned limitation of the context size [9], we use *Instcombine* in our study to train LLMs.

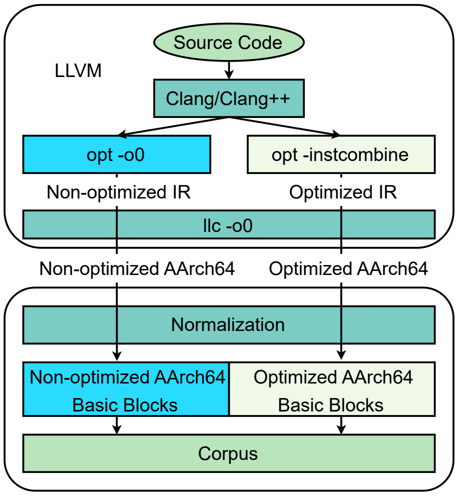


Fig. 1: Basic block generation pipeline.

Generation of the training data: To investigate whether LLMs can learn how to apply peephole, we use the LLVM compiler to generate optimized and non-optimized versions of basic blocks. Figure 1 shows the code generation pipeline used in our study. We first compile the source code into IR using the `clang/clang++` frontend. The non-optimized IR is generated by invoking `opt` with the `-O0` flag. For generating the optimized IR, we use the `opt -instcombine` command to perform a series of peephole optimizations. These IRs are compiled into non-optimized and optimized AArch64 assembly code using `llc` with the `-O0` flag. Finally, we parse the generated AArch64 assembly codes and match optimized basic blocks with non-optimized ones. We also apply the following pre-processing steps (normalization):

- Remove metadata and attributes.
- Remove basic blocks with more than 15 lines of instructions[2].
- Remove duplicate basic blocks with identical instructions.

We apply these pre-processing steps to follow previous studies [9].

The LLM-based optimization pipeline: Figure 2 demonstrates the LLM-based code optimization pipeline. We first compile the source code and generate AArch64 assembly. Then we apply the pre-processing steps discussed previously to each basic block.

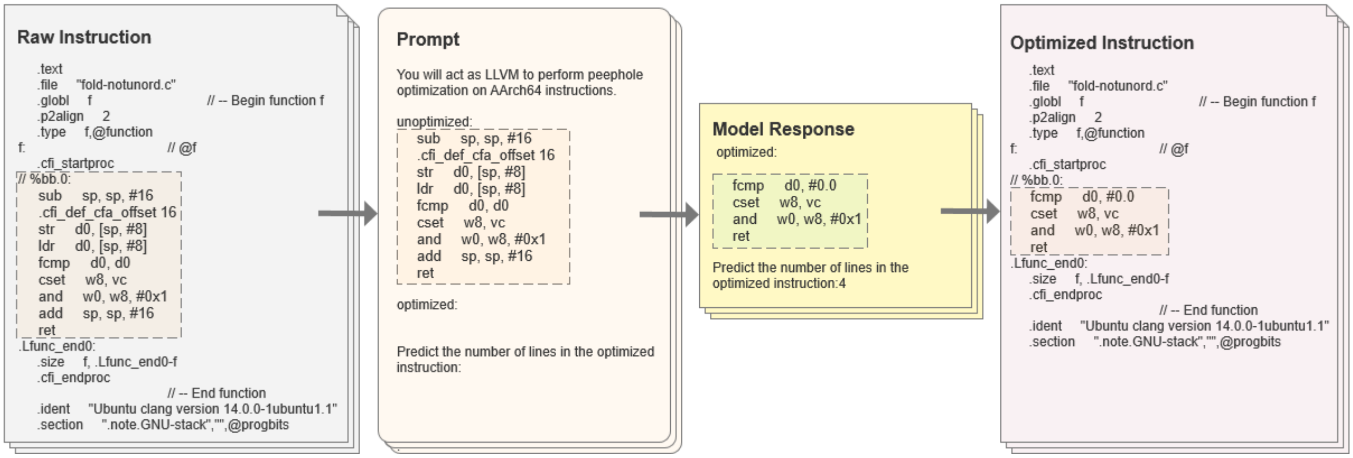
We embed each basic block into a specific prompt template to guide LLMs in performing peephole optimization. Once a model generates the optimized basic block, we replace the original basic block with the optimized one and generate a binary[3].

---

[1] We provide examples of typical optimizations performed by peephole in Table VIII for the readers who are not familiar with compilers.

[2] We limit the size of basic blocks due to constraints of our computing facilities. Additionally, access to the OpenAI models used in our study is a paid service, with costs incurred based on the number of tokens processed.

[3] Note that there are limitations of such an approach which we discuss in Section V.



```
        .text
        .file    "fold-notunord.c"
        .globl   f                          // -- Begin function f
        .p2align    2
        .type    f,@function
f:                                  // @f
        .cfi_startproc
// %bb.0:
        sub      sp, sp, #16
        .cfi_def_cfa_offset 16
        str     d0, [sp, #8]
        ldr     d0, [sp, #8]
        fcmp      d0, d0
        cset     w8, vc
        and      w0, w8, #0x1
        add      sp, sp, #16
        ret
.Lfunc_end0:
        .size    f, .Lfunc_end0-f
        .cfi_endproc
                                    // -- End function
        .ident    "Ubuntu clang version 14.0.0-1ubuntu1.1"
        .section     ".note.GNU-stack","",@progbits
```

You will act as LLVM to perform peephole optimization on AArch64 instructions.

```
unoptimized:
    sub     sp, sp, #16
    .cfi_def_cfa_offset 16
    str    d0, [sp, #8]
    ldr    d0, [sp, #8]
    fcmp     d0, d0
    cset    w8, vc
    and     w0, w8, #0x1
    add     sp, sp, #16
    ret

optimized:
```

Predict the number of lines in the optimized instruction:

```
optimized:

    fcmp     d0, #0.0
    cset    w8, vc
    and     w0, w8, #0x1
    ret
```

Predict the number of lines in the optimized instruction:4

```
        .text
        .file    "fold-notunord.c"
        .globl   f                          // -- Begin function f
        .p2align    2
        .type    f,@function
f:                                  // @f
        .cfi_startproc
// %bb.0:
        fcmp      d0, #0.0
        cset     w8, vc
        and      w0, w8, #0x1
        ret
.Lfunc_end0:
        .size    f, .Lfunc_end0-f
        .cfi_endproc
                                    // -- End function
        .ident    "Ubuntu clang version 14.0.0-1ubuntu1.1"
        .section     ".note.GNU-stack","",@progbits
```

Fig. 2: The code optimization pipeline.

LLMs: In this study, we use the 7B-parameter Llama2 model as a baseline model, which is widely recognized for outperforming many other open-source models across multiple benchmarks [16].

Apart from the Llama2 model, we use pre-trained OpenAI GPT-4o [17] and GPT-o1 (preview) [18] models to investigate code optimization efficiency for LLMs with enhanced reasoning logic. The difference between these models is that GPT-o1 provides mode advanced structured reasoning and problem-solving capabilities [16].

TABLE I: Examples demonstrating the Llama2 model's understanding of AArch64 instructions (without fine-tuning) are shown. Cases (a) and (b) demonstrate that Llama2 accurately interprets the instructions, while case (c) illustrates a more complex scenario where Llama2 fails to correctly comprehend the instructions.

Prompt: What does `ldr` mean in AArch64 instructions?

Response: The `ldr` instruction loads a 64-bit value from memory into a 64-bit register.

(a) This response is correct.

Prompt: What does `sub` mean in AArch64 instructions?

Response: A `sub` instruction in AArch64 is an instruction that performs a subtraction operation on two 64-bit operands.

(b) This response is correct.

Prompt: What does `sxtw` mean in AArch64 instructions?

Response: In AArch64 (64-bit ARM) instructions,`sxtw` is a mnemonic that stands for "shift extract with write"

(c) Incorrect. The `sxtw` stands for "Sign Extend Word."

Fine-tuning of the baseline model: Llama2 is pre-trained on a large corpus, with 4.5% of the data sourced from public code on GitHub [10]. We directly queried the model to assess its understanding of the specific meanings of AArch64 instructions. Table I presents three distinct questions related to AArch64 instructions. While the model demonstrates an ability to understand some instructions, it does not recognize more complex instructions.

To explore this further, we provided the model with several prompt templates (shot learning prompts) prior to submitting optimization requests to guide its understanding. However, our results indicate that the model was still unable to perform the requested optimizations. To enhance accuracy of the Llama2 model, we apply a fine-tuning process [19].

In our study, we also use pre-trained OpenAI GPT-4o and GPT-o1 models. However, we cannot fine-tune these models since we do not have access to source code.

Quantization of the baseline model: A complete fine-tuning of LLMs is very expensive. When using the regular AdamW optimizer [20], the Llama2-7B model requires a minimum of 8 bytes per parameter for its 7 billion parameters during training, totaling 56 GB of GPU memory[4]. To reduce memory usage, we employ QLoRA (Efficient Fine-tuning of Quantized LLMs) [21], which significantly decreases the parameter storage requirements.

QLoRA applies the following major three techniques:

1) 4-bit NormalFloat Quantization: This reduces the precision of model parameters to 4 bits while maintaining numerical stability.
2) Dual Quantization: This technique further reduces average memory usage by quantizing the quantization constants themselves.
3) Unified Memory Paging Technology [22]: This leverages NVIDIA's unified memory mechanism to enhance the efficiency of memory page transfers between CPU and GPU, thereby managing memory peak values.

---

[4] In our experimental framework, GPUs have only 40Gb of DRAM.

We applied low-rank adaptation with a rank of 8 to the query and value projection layers of Llama2 and used the standard QLoRA configuration. This technique reduces the trainable parameter number by 1606x, retaining approximately 0.06% of the original parameters.

Optimization of the baseline model: The model, with 4,194,304 trainable parameters, is trained using the Paged AdamW optimizer [20] with a learning rate of $2 * 10^{-4}$ and a constant learning rate schedule. The total training batch size is 128, achieved through micro-batches of 32 and gradient accumulation steps of 4. We trained the model on an Intel-based Cloud server using NVidia A100 with 40Gb of internal DRAM (VRAM).

III. EVALUATION OF THE BASELINE MODEL

In this section, we present the evaluation results for our baseline model and compare our results with the previous studies. We aim to replicate the training and evaluation strategy used in a recently presented study [9].

A. *Model Evaluation Metrics*

To assess the performance of the baseline model, we use conservative natural language evaluation metrics. These metrics provide a quantitative measure of how closely the code optimized by LLMs aligns with the code optimized by the compiler. In addition to these metrics, we also use metrics to evaluate the syntactic correctness and the accuracy of the IO output for code generated by LLMs.

BLEU Score: The BLEU (Bilingual Evaluation Understudy) score [23] is commonly used in machine translation to evaluate the similarity between machine-generated sentences and reference sentences. In our context, it measures how closely the basic blocks optimized by LLMs match the basic blocks optimized by the compiler.

Exact Match Rate (EMR): This metric also enables us to compare the code samples generated by LLMs and code samples generated by the compiler. To be more specific, it measures the character-to-character complete matching rate for the basic blocks generated by LLMs and the basic blocks generated by the compiler. In other words, EMR only gives credit if the entire generated output exactly matches the reference output, making it a more binary metric than BLEU; an exact match implies a BLEU score of 1.

Syntactic Accuracy: The syntactic accuracy measures the proportion of basic blocks generated by LLMs that can be correctly compiled without errors by LLVM. This metric ensures that the generated instructions are syntactically correct.

IO Accuracy: IO accuracy is used to check whether, for a given set of inputs, the code generated by LLMs produces the same output as the code optimized by the compiler. IO accuracy has been validated as an effective approach in prior research on code generation [5].

Importantly, we use these metrics to follow the previous studies exploiting LLMs for code generation [5], [9], enabling us to verify our results.

B. *Fine-tuning*

To tune the baseline model, we use C and C++ code samples from the LLVM and GCC test suites [24].

We specifically use these suites since both suites were designed to test the code generation and optimization functionality of the compilers. Thus, we expect that these test suites ideally fit for the purposes of fine-tuning covering the majority of possible code optimization scenarios.

We extracted 100,000 basic blocks (27 million tokens in total) from the test suites to fine-tune the Llama2 model. The code generation pipeline (see Figure 1) is applied to create both optimized and non-optimized basic block samples. We found that 6 epochs is optimal for our study, as increasing the number of epochs yields minimal changes in the target metrics, i.e. BLEU, EMR, Syntactic Accuracy and IO Accuracy. Note that the context window size is 512 tokens.

C. *Evaluation using the LLVM/GCC test suite*

Similar to the previous study [9], we first evaluate the model throughout the fine-tuning process. To be more specific, after every 10,000 samples used for fine-tuning we assess the model on a holdout validation set. This set consist of 5000 basic block samples that have not been used for training.

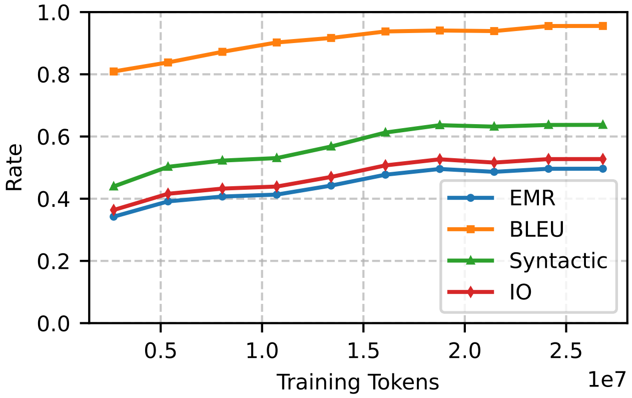


Fig. 3: Change in target LLM accuracy metrics with the number of tokens used for fine-tuning.

Figure 3 demonstrates how the target LLM accuracy metrics change if we increase the size of the data (the number of tokens) used for fine-tuning. We see that BLEU achieves a score of 0.92, indicating that the LLM generates the code which is quite similar to the code generated by LLVM. However, EMR achieves only 0.5 which implies that the model does not fully replicate the behavior of the compiler. Syntactic Accuracy and IO Accuracy reach 0.66 and 0.54, respectively.

A previous study, which conducted similar experiments with the Llama2 model [9], has demonstrated results comparable to ours but slightly higher, i.e. BLEU at 0.952, EMR at 0.6 and Syntactic accuracy at 0.87. However, in this study, authors have trained LLMs to mimic the behavior of multiple compiler optimizations in order to reduce the size of the code. Moreover, authors trained the Llama2 model from scratch, while we performed fine-tuning. Finally, the previous study used much larger dataset, consisting of 1,000,000 IR functions and totaling approximately 10.9 billion tokens, compared to 27 million tokens in our study.

D. Evaluation using Code-Force, Code-Jam and BigCode

In this subsection, we present the results of the performance evaluation of the baseline model using a large dataset that combines thousands of algorithms implemented on major coding competition platforms. Our dataset is derived from two algorithmic competitions: Code-Force and Google Code Jam, with a particular focus on solutions written in C and C++. We have extracted the source code from GitHub applying pre-processing steps to remove duplicate code samples. In addition, we employed the BigCode's The Stack public repository, which is dedicated to training large language models with code [25].

To extract basic blocks, we have processed about 300,000 C/C++ source code files, but we were able to compile and generate binaries only for 106,082 files. [5] The detailed information about the number of extracted source code files and basic blocks is provided in Table II. Overall, we have managed to extract 437,892 basic blocks for which we can generate the binaries. Due to limitations in our computing resources, we randomly select 24,000 basic blocks to evaluate the performance of the fine-tuned Llama2 model.

TABLE II: Testing dataset info.

	Source Files	**Basic Blocks**	**Sampled Blocks**
Code-Force	1,050	5,610	4,000
Code-Jam	10,764	110,354	10,000
BigCode	94,268	321,928	10,000
Total	106,082	437,892	24,000

E. Results

Figure 4 demonstrates BLEU, EMR, Syntactic accuracy and IO accuracy measured for 24,000 basic blocks optimized by the fine-tuned Llama2 model. Our first observation is that BLEU is almost the same for all the datasets, about 0.87, and it is slightly lower than BLEU measured in our previous experiments. However, EMR drops significantly compared to the previous experiments, and it varies from 0.27 to 0.39. This is explained by the fact that the training set significantly differs from the large testing set, and, as a result, the LLM cannot predict accurately the output generated by the compiler. However, we see only slight degradation of Syntactic Accuracy and IO accuracy drops compared to the previous experiments. These results suggest that the LLM has some generalization capabilities, as EMR decreases while the quality of the generated code, in terms of Syntactic Accuracy and IO Accuracy, remains almost the same.

Importantly, these results are aligned with the results obtained in the previous studies [5], [9]. For example, it is shown that the model translating C code into x86 instructions achieves a BLEU score of 0.79, a Syntactic Accuracy of 0.58, and an IO Accuracy of 0.33 [5].

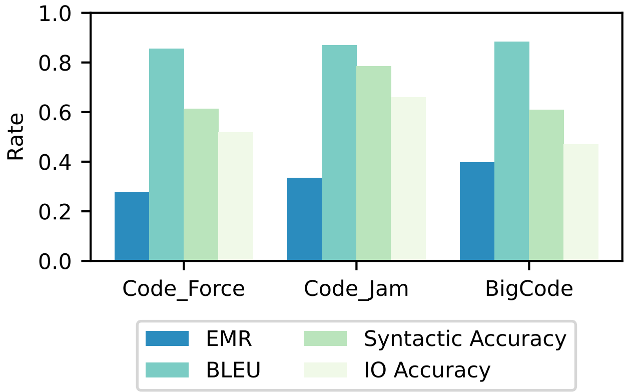


Fig. 4: Model performance on different test sets.

F. Error analysis

Next, we conduct a systematic analysis of errors produced by the LLM. The model produced a total of 8,922 compilation errors. Note that we focus specifically on compilation errors, as they are easier to identify the sources of. We randomly select 1,000 of these errors to make a systematic analysis. Table IIIa shows the distribution of 1,000 compilation errors across four different error types which we discuss below.

Opcode errors refer to invalid instruction codes, invalid instruction names, or inappropriate instruction names for a particular use generated by the LLM. An example of such errors is provided in Table V. In this particular case, the LLM generated *and* instruction instead of *mov* instruction. Note that this type of error occurs most frequently in our experiments (see Table IIIa).

Immediate Value Errors are related to the errors introduced by the LLM when using literal constants.

For example, Table V shows incorrect usage of literal constants. To be more specific, *mov* instruction should initialize register *w0* with 0 using register *wzr*. However, it generates *#r* instead of *wzr*. Note that # denotes an immediate value, and *r* should be a literal constant in this particular case.

Label Errors represent the errors due to incorrect labels and function names in assembly code. For example, in Table V, *adpr* instruction should load the address of *.L.str* label. However, the LLM generated an incorrect label, i.e. *.Lstrstr*.

Register Errors represent the errors induced by the LLM due to incorrect usage of registers. For example, in Table V *mov* instruction uses 32-bit register *w8* instead of 64-bit register *x8*.

Next, we further investigate the errors that occurred most frequently in our experiments, i.e. opcode errors. We estimated

---

[5] The C/C++ files extracted from these datasets often include external dependencies, such as third-party libraries or project-specific headers. However, these header files and external libraries are not included in the datasets, which limits our ability to generate binaries. As a result, we were only able to obtain binaries for source files that use standard libraries.

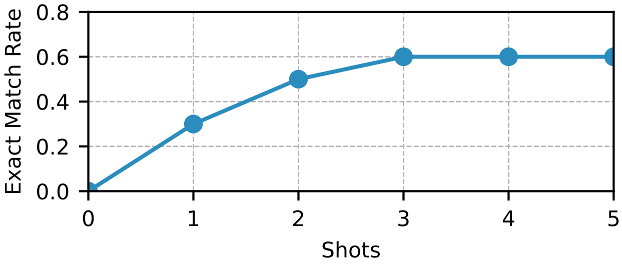


Fig. 5: Correlation between EMR (Exact Match Rate) and the number of prompt shots.

the probability of opcode errors for each instruction obtained in our testing dataset. Table IIIb and IIIc show top 10 instructions with the highest and lowest error probabilities, including 95% confidence intervals, respectively. Note that these tables contain only those instructions which have a representative, i.e. a statistically significant, number of samples in the training dataset, i.e. above 50.

We observe that the error probability varies from 0.5% for the *stp* instruction to 43% for the *eor* instruction.The list of most error-prone errors include:

- *eor (Bitwise exclusive-OR)*. The LLM often confuses this instruction with *sub*, using it in place of *eor*. It also confuses *eor* with *and*, *mov* and *ls*;
- *ldurb* (Load Register byte). The model confuses unsigned byte loads (*ldurb*) with signed loads, such as *ldrsb* (Load Register Signed Byte) and *ldrsh* (Load Register Signed Half-word);
- *asr* (Arithmetic Shift Right). The LLM tends to confuse this instruction with the logical shift instruction. Additionally, it often generates invalid instruction names, such as *movr* and *as+sr*.
- *adds*. The LLM struggles to differentiate between add instructions with and without flag setting, i.e. (*add*) and (*adds*). It also occasionally generates incorrect instructions, such as *sub*, *strs*, *subs* and *mov* instructions.
- *lsr*. The model often generates the non-existent *movr* instruction, which is not part of Aarch64 ISA.
- *stur*. LLM generates various non-existent instructions, such as *strur*, *blur*, *tbur*, *movur*.

Our results indicate that, in the vast majority of error cases, the LLM model generated non-existent instruction names.

One possible explanation for this could be the lack of sufficient samples for certain instructions in the training dataset. However, even though *stur* is one of the most frequently sampled instructions in the dataset (with 21,277 samples, see Table IV), it still ranks among the top 10 instructions with the highest error probability.

Interestingly, the *stp* instruction has the lowest error probability (see Table IIIc). However, in a few instances, the LLM generates non-existent *subp* instruction instead of *stp*. We attribute this to the frequent occurance of both *stp* and *sub* in the beginning of functions, leading the model to confuse these instructions and generate *subp*. Such errors might be avoided if the LLM was guided by a reasoning-based approach rather than relying solely on the context.

Fundamental limitation of conventional LLMs for code generation. Overall, our findings indicate that no instructions have an error probability of zero[a], and the vast majority of errors are due to incorrectly generated instruction names. In other words, conventional LLMs exhibit some probability of error when generating opcodes for every instruction, and this probability is always greater than zero. We attribute this to the probabilistic nature of LLMs' output, which significantly limits the use of conventional LLMs for code generation. All the observations and errors, especially opcode errors and register errors, presented above highlight the need for LLMs to incorporate a reasoning mechanism and enhanced logic capabilities to overcome this fundamental limitation.

[a] The number of samples for which is representative, i.e. larger than 50.

IV. EVALUATION OF LLMS WITH ADVANCED REASONING MECHANISMS

To investigate if LLMs can increase the rate of correct transformation by enabling reasoning mechanisms, we tested OpenAI GPT-4o and GPT-o1 (preview) which have advanced reasoning capabilities [18].

A. The fine-tuned Llama2 vs GPT-4o

We first evaluated OpenAI's GPT-4o model using our Code-Force, Code-Jam and BigCode datasets, as discussed in Section III-D. Since we cannot fine-tune GPT-4o (the source code is not available), we use 3-shot context learning prompts [6] before requesting the model to optimize the code following the same pipeline as for the Llama2 model, see Figure 2. Note that for all the optimization requests we use absolutely the same 3 learning prompts.

To motivate the use of prompt-based training, we conducted a series of experiments optimizing 20 randomly chosen basic blocks. We also selected 5 pairs of optimized and non-optimized basic blocks, which are randomly sampled, and use them as prompts. Figure 5 shows how the average EMR (the exact match rate for basic blocks generated by the LLM and blocks optimized by the compiler), measured across 20 basic blocks, changes with the number of prompts. We see that EMR grows more than 12x after applying the 3-shot prompt learning, and it reaches EMR about 60%.

Note that EMR does not change when the number of shot prompts has increased above 3. Based on these results, we use the 3-shot prompt learning to evaluate GPT-4o using 24,000 basic blocks extracted from Code-Force, Code-Jam and BigCode datasets.

We compare the fine-tuned Llama2 and GPT-4o using four key metrics: BLEU, EMR, IO accuracy, and Syntactic accuracy. Figure 6 shows the evaluation results. We see that

---

[6] 3-shot learning implies that before requesting the models to optimize code, these models are provided 3 examples of the optimized and non-optimized code samples.

TABLE III: Distribution of compilation errors for Code-Force, Code-Jam and BigCore datasets.

(a) Distribution of 1000 compilation errors across 4 different types of errors.

Error Category	Count
Opcode Error	526
Immediate Value Error	518
Label Error	272
Register Error	188
Total	**1000***

*The total number of errors does not match 1000 errors since some code samples have multiple errors.

(b) Opcode errors: Top 10 instruction types with the highest error probability; the number of samples > 50.

Instr	Error Count	Total Count	Error Prob	Conf
eor	22	51	0.431373	0.135929
ldurb	73	175	0.417143	0.073057
asr	29	81	0.358025	0.104407
adds	22	65	0.338462	0.115035
lsr	24	72	0.333333	0.108889
fmov	58	222	0.261261	0.057791
stur	1381	5718	0.241518	0.011094
tbnz	140	588	0.238095	0.034427
ldursw	35	151	0.231788	0.067306
orr	16	71	0.225352	0.097187

(c) Opcode errors: Top 10 instruction types with the lowest error probability; the number of samples > 50.

Instr	Error Count	Total Count	Error Prob	Conf
stp	37	7250	0.005103	0.001640
ldp	37	5164	0.007165	0.002300
ret	73	7028	0.010387	0.002370
fadd	9	682	0.013196	0.008565
bl	218	15087	0.014450	0.001904
cbz	27	1145	0.023581	0.008789
b	499	18584	0.026851	0.002324
mul	37	1040	0.035577	0.011258
sdiv	46	1192	0.038591	0.010935
fcmp	7	181	0.038674	0.028091

TABLE IV: Top 20 instructions with the most samples in the training dataset.

Instr	Count	Instr	Count	Instr	Count
b	107630	subs	28560	cbz	6880
ldr	78803	stur	21277	mul	6231
str	60305	adrp	12884	cbnz	5616
mov	46248	sub	11865	ldp	5152
add	42832	ret	10814	and	4553
ldur	39714	stp	9103	strb	4051
bl	30585	udiv	8420		

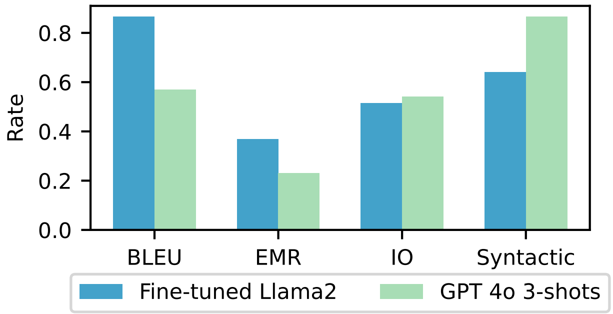


Fig. 6: Average BLEU, EMR, IO accuracy, Syntactic accuracy obtained for 24,000 basic block samples (Code-Force, Code-Jam and BigCode datasets).

the fine-tuned Llama2 outperforms GPT-4o in terms of BLEU and EMR metrics. This can be explained by the fact that the Llama2 model was specifically fine-tuned on peephole optimization code samples, aligning the code generated by the model with that produced by the specific compiler used in our study. Nonetheless, GPT-4o demonstrates a higher Syntactic accuracy and IO accuracy in our experiments which implies that this model has better capabilities to generate syntactically and semantically correct code. We attribute this to the fact that GPT-4o has a larger number of parameters and was trained on a diverse data.

Finding: Based on these results, we conclude that BLEU and EMR metrics widely used in NLP [26], [27]studies do not really serve the purposes of code quality evaluation. Thus, the use of these metrics should be reconsidered in future research studies on code generation.

B. Llama2 vs GPT-4o vs GPT-o1

In our final experiments, we tested the performance of an OpenAI model with the most advanced reasoning capabilities, i.e. GPT-o1 (preview)[7]. Unfortunately, GPT-o1 is a paid model, which limits the number of code samples we could test in our study. To address this, we selected 200 most complicated code samples from 24,000 samples (Code-Force, Code-Jam and BigCode datasets) used for testing the fine-tuned Llama2 and GPT-4o. Note that we selected these samples in a such a way that the fine-tuned Llama2 model generates the code which differs from the the code produced by LLVM and produce errors in 190 samples; and we chosen 10 samples where both Llama2 and GPT-4o generate the correct code which is similar to the code generated by LLVM (200 samples in total). Such a division of the samples enables us to test GPT-o1 with both types of block samples, those correctly and those incorrectly optimized by Llama2.

Table VI shows the evaluation results for all models tested with 24,000 samples (*ALL*) and 200 selected samples (*Selected*).

---

[7] https://openai.com/index/introducing-openai-o1-preview/

TABLE V: Examples of errors introduced by Llama2.

Error Type	**Incorrect transformation**	**Correct transformation**
Opcode Error	mov w8, w0, #0xff\n lsr w8, w8, #4\n orr w0, w8, w9	and w8, w0, #0xff\n lsr w8, w8, #4\n orr w0, w8, w9
Immediate Value Error	mov w0, #r\n ret	mov w0, wzr\n ret
Label Error	adrp x0, .Lstrstr\n add x0, x0, :lo12:.L.str	adrp x0, .L.str\n add x0, x0, :lo12:.L.str
Register Error	mov w8, x0\n mov w0, #3\n str w0, [x8]\n ret	mov x8, x0\n mov w0, #3\n str w0, [x8]\n ret

TABLE VI: The performance of LLMs.

Test Samples	**Metrics(%)**	**Llama2 7B(Fine-tuned)**	**GPT-4o (3-shots)**	**GPT-o1 (3-shots)**
ALL	EMR (Exact Match Rate)	36.8	23.1	-
	Syntactic Accuracy	64.0	86.6	-
	IO Accuracy	51.6	54.2	-
	BLEU	86.6	56.9	-
Selected	EMR (Exact Match Rate)	5.0	5.0	19.0
	Syntactic Accuracy	12.5	68.0	92.0
	IO Accuracy	7.3	49.1	79.1
	BLEU	73.0	35.0	78.0

We see that GPT-o1 outperforms both GPT-4o and the fine-tuned Llama2 across three metrics for the 200 selected samples, i.e. EMR, IO accuracy, Syntactic accuracy. The fine-tuned Llama2 model shows the lowest Syntactic accuracy and IO accuracy for the selected samples. We believe this is due to the model's lack of generalization capabilities, as we found that if the Llama2 model generates code that differs from the one produced by LLVM, it is highly likely to result in a compilation error, program crash, or incorrect output. Meanwhile, GPT-4o and GPT-o1, which were not specifically fine-tuned on code samples with applied peephole optimization, often generate the correct code, although it does not match the code generated by LLVM.

Finding: Based on these observations, we conclude that fine-tuning tends to adjust LLMs to completely mimic the behavior of compilers, which may negatively affect their generalization capabilities and overall performance.

Importantly, although both GPT-4o and GPT-o1 have not been fine-tuned, GPT-o1 significantly outperforms GPT-4o. To be more specific, GPT-o1 achieves 14% higher EMR, 24% higher Syntactic Accuracy, and 30% higher IO Accuracy compared to GPT-4o. We attribute such a difference to the enhanced reasoning capabilities of GPT-o1.

To explore this, we made a deep investigation of a particular code sample for which both GPT-4o and the Llama2 model generate incorrect code, while GPT-o1 successfully produce the correct code. Table VII shows the original code sample and code samples optimized by LLVM, Llama2, GPT-4o and GPT-o1. The original code sample stores the value 5 at a 64-bit address contained in the *x0* register with an offset based on the value in the 32-bit *w1* register, shifted left by 2 and extended to 64 bits. However, this code contains several redundant instructions, including instructions writing and reading from the stack.

We see that GPT-4o made several errors. Specifically, it included a redundant move instruction (*mov w8, w1*) and generated an incorrect load instruction which reads data from a stack address (*[sp, #12]*), while the model removed the store instruction which writes to this address. Thus, GPT-4o does not have enough reasoning capabilities to understand the optimization logic and avoid such simple errors. Nonetheless, the model generated syntactically correct code which can be compiled. Meanwhile, the Llama2 model generated syntactically incorrect code. Specifically, it generated incorrect opcodes, such as *movsl* and *movr*; the second argument for *str w8, #5* instruction also does not follow the assembly syntax. We explain this, as discussed previously, by the fact that fine-tuning adjust the model to mimic the behavior of the compiler which negatively affects the model generalization ability, especially for those code samples which significantly differ from samples used for fine-tuning.

Finally, GPT-o1 correctly optimized the code. Moreover, we see that GPT-o1 generated better code than LLVM since it used only three instructions, while LLVM used five instructions. In particular, both GPT-o1 and LLVM removed instructions which allocate space in stack and use it which are redundant in this case. However, LLVM generated redundant *lsl* and *asr* instructions, while GPT-o1 performed the offset transformation by adding *sxtw #2* to *str* instruction. Thus, we see that GPT-o1, with enhanced reasoning mechanisms, can outperform not only LLMs lacking advanced reasoning capabilities but also the compilers [8].

Reasoning and chain-of-thought. We believe that the high performance of GPT-o1 compared to Llama2 and GPT-4o is primarily due to the implemented chain-of-thought mechanism. The key advantage of this mechanism is that the model makes several prompts to optimize the code in several stages, replicating the reasoning process. For example,

---

[8] Note that the basic block represents an entire function in this case. Thus, there are no other basic blocks in this function which contain uses in the data-flow graph for the removed instructions. We also verified that the generated code executes correctly.

Figure 8 shows the chain-of-thought applied by GPT-o1 to optimize the presented above code sample. We see that it is trying to understand the overall purpose of the provided code sample and examine the function flow. Moreover, it also builds a pseudo code for the provided code sample which is further used to generate the optimal code.

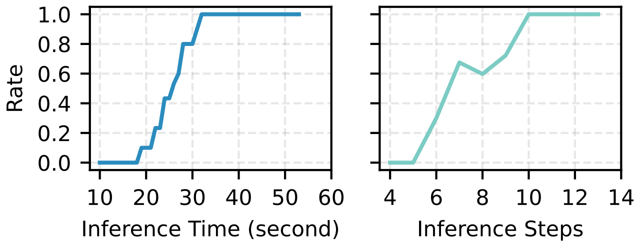


Fig. 7: The rate at which GPT-o1 produces the most optimal code, inference time and the number of inference steps.

But why is it specifically due to the chain-of-thought process? Indeed, there are rumors that GPT-o1 implements more parameters than other models, which might allow it to outperform others due its size rather than chain-of-thought. To investigate this, we made several manipulations to vary time and the number of inference steps within the chain-of-thought process for a particular basic block optimization presented in Table VII. We made 40 experiments adjusting the number of inference steps and inference time, and measured the rate at which GPT-o1 produces the most optimal code, i.e. the code presented in Table VIIe. Figure 7 illustrates how this rate changes with the number of inference steps and inference time. The results clearly show that the rate grows with both inference time and the number of inference steps. Notably, GPT-o1 generates the most optimal code in 100% cases when the inference time exceeds 34 seconds or the number of inference steps is greater than 10 steps. In contrast, GPT-o1 never generates the optimal code when inference time is under 17 seconds or the number of steps is below 5. Furthermore, it also produces syntactically incorrect code in these cases. These findings suggest that the chain-of-thought mechanism significantly impacts GPT-o1's performance and the quality of the generated code.

Overall, we see that the chain-of-thought mechanism implemented by GPT-o1 significantly improves the performance of the model in generating optimized syntactically and functionally correct code. We believe that this is due to the fact that the model tries to understand the code and apply optimizations systematically. However, a better understanding the performance of LLMs with enhanced reasoning mechanisms for code generation and optimization requires a further research with open-sourced models.

V. LIMITATIONS AND FUTURE RESEARCH.

There are several limitations in our study which we believe are important to discuss.

Def-use graph across different basic blocks. In our study, LLMs optimize each basic block separately. Note that we

TABLE VII: An example of code sample and peephole transformations performed by LLVM, Llama2, GPT-4o and GPT-o1.

(a) Original Code

```
sub sp, sp, #16
.cfi_def_cfa_offset 16
mov w8, w1
str w8, [sp, #12]
str x0, [sp]
ldr x9, [sp] // Load x9 with value at [sp] (x0)
ldrsw x10, [sp, #12] // Load signed word from [sp,
    ↪  #12] into x10, sign-extended to 64 bits
mov w8, #5 // Move immediate 5 into w8
str w8, [x9, x10, lsl #2] // Store w8 at [x9 + (
    ↪ x10 << 2)]
add sp, sp, #16
ret
```

(b) Peephole performed by LLVM

```
lsl x8, x1, #32 // Logical shift left x1 by 32
    ↪ bits, result in x8
asr x9, x8, #32 // Arithmetic shift right x8 by 32
    ↪  bits, sign-extend into x9
mov w8, #5 // Move immediate 5 into w8
str w8, [x0, x9, lsl #2]
ret
```

(c) GPT-4o Output

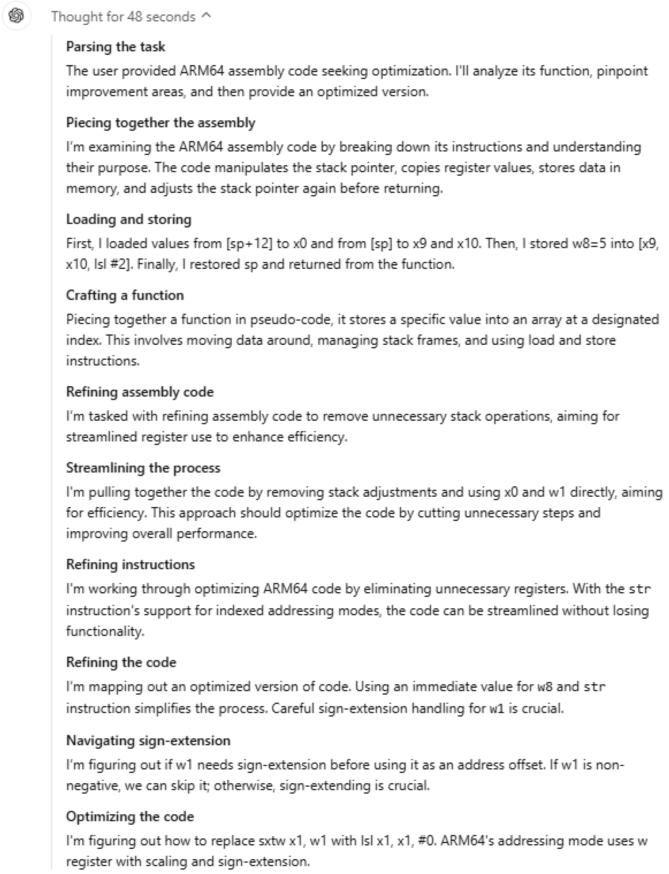
```
mov w8, w1
mov w9, x0
ldrsw x10, [sp, #12] //
    ↪  Attempts to
    ↪ load from [sp,
    ↪ #12], but no
    ↪ valid data there
mov w8, #5
str w8, [x9, x10, lsl
    ↪ #2]
ret
```

(d) Llama2 Output

```
movsl x8, x0, #2 //
    ↪ Invalid opcode ’
    ↪ movsl’
movr x8, x0, #2 //
    ↪ Invalid opcode ’
    ↪ movr’
str w8, #5 // Incorrect
    ↪  syntax; ’str’
    ↪ requires a
    ↪ memory address
str w8, [x0, x9, lsl
    ↪ #2]
ret
```

(e) GPT-o1 Output

```
mov w8, #5
str w8, [x0, w1, sxtw #2] // Store w8 at [x0 + (
    ↪ SignExtend64(w1) << 2)]
ret
```



Parsing the task

The user provided ARM64 assembly code seeking optimization. I'll analyze its function, pinpoint improvement areas, and then provide an optimized version.

Piecing together the assembly

I'm examining the ARM64 assembly code by breaking down its instructions and understanding their purpose. The code manipulates the stack pointer, copies register values, stores data in memory, and adjusts the stack pointer again before returning.

Loading and storing

First, I loaded values from [sp+12] to x0 and from [sp] to x9 and x10. Then, I stored w8=5 into [x9, x10, lsl #2]. Finally, I restored sp and returned from the function.

Crafting a function

Piecing together a function in pseudo-code, it stores a specific value into an array at a designated index. This involves moving data around, managing stack frames, and using load and store instructions.

Refining assembly code

I'm tasked with refining assembly code to remove unnecessary stack operations, aiming for streamlined register use to enhance efficiency.

Streamlining the process

I'm pulling together the code by removing stack adjustments and using x0 and w1 directly, aiming for efficiency. This approach should optimize the code by cutting unnecessary steps and improving overall performance.

Refining instructions

I'm working through optimizing ARM64 code by eliminating unnecessary registers. With the `str` instruction's support for indexed addressing modes, the code can be streamlined without losing functionality.

Refining the code

I'm mapping out an optimized version of code. Using an immediate value for `w8` and `str` instruction simplifies the process. Careful sign-extension handling for `w1` is crucial.

Navigating sign-extension

I'm figuring out if w1 needs sign-extension before using it as an address offset. If w1 is non-negative, we can skip it; otherwise, sign-extending is crucial.

Optimizing the code

I'm figuring out how to replace sxtw x1, w1 with lsl x1, x1, #0. ARM64's addressing mode uses w register with scaling and sign-extension.

Fig. 8: A chain-of-thought produced by GPT-o1.

specifically use *Instcombine* optimization which is applied to each basic block without additional transformation of CFG. However, removed instructions can have uses in other basic blocks. Thus, LLMs can produce instructions which arguments are initialized incorrectly. Nonetheless, we expect that LLMs should account for this (their advanced reasoning mechanisms should enable it) by defining the result registers for removed instructions when those registers might be used in other basic blocks. Note that the instructions without replacement can be removed only in the terminal CFG nodes, i.e. containing *ret* as in the example from Table VII.

Register allocation. Since LLMs only see the instructions within an optimized basic block, they can use only the registers that are used within that block. This is due to the fact that LLMs do not have the information about liveliness of all the registers.

The entire program representation. Overall, we admit that that there could be also other limitations due to the fact that LLMs do not see the entire program code. Moreover, in certain cases, runtime support may be required, similar to what is implemented in binary translators. However, we note that the main goal of our study was to investigate whether LLMs have sufficient capabilities to perform a simple peephole optimization and whether enhanced reasoning mechanisms can improve the models' performance.

In our future research, we will investigate the optimization capabilities of LLMs using a full program representation.

VI. RELATED WORK

Large Language Models (LLMs) have shown their efficiency in performing various tasks ranging from high-level code generation to compiler optimization and advanced reasoning.

Large Language Models for Code Generation. In recent years, there has been a growing interest in leveraging Large Language Models (LLMs) for tasks involving source code generation. Models such as Copilot [1], Codex [2], TransCoder [3], CodeBERT [4], Code Llama [28], StarCoder [25], [29], Magicoder [30] and DeepSeek-Coder [31] have significantly advanced this field. These models support developers with tasks like code completion, generation, and translation across multiple programming languages. Open-sourced models like Code Llama and StarCoder have further empowered the community to adapt and fine-tune LLMs for specific software engineering needs.

Large Language Models for Compilers. While LLMs have demonstrated significant potential in high-level code tasks, fewer models operate at the compiler level, particularly with code generation and compiler optimization. Most recent studies have focused on traditional machine learning methods for compiler optimization [32]–[38]. Neural machine tranlation techiques have been employed to transform code between different representations, previous examples include compiling C to X86 assembly [5] and decompiling assembly language to C [6], [7]. These works utilized smaller models or other deep learning methods. There are a few works related to using LLM at the compiler level. Examples include using large models for decompilers [39]–[41], LLVM-IR passes prediction with IR optimization [9], compiler fuzzing tests [42], [43].

Recent Developments in Reasoning. Enhancing the reasoning capabilities of LLMs has been a focus of recent research, particularly through methods like chain-of-thought (CoT) prompting. CoT encourages models to generate intermediate reasoning steps, improving performance on complex tasks such as mathematical reasoning and question answering [44]. Chen [45] proposed the chain-of-code (CoC) method. This approach enables models to handle tasks that require both logical and semantic reasoning, broadening the scope of questions LLMs can answer by ”thinking in code”. Experiments demonstrated that chain-of-code outperforms chain-of-thought and other baselines across various benchmarks. Gu et al. [46] introduced CRUXEval (Code Reasoning, Understanding, and Execution Evaluation), a benchmark specifically designed to test input-output reasoning in code. Wang [47] introduced OpenR, an open-source framework designed to enhance the reasoning capabilities of LLMs through reinforcement learning and process supervision. OpenR achieves advanced reasoning capabilities beyond traditional auto-regressive methods. To the best of our knowledge, there are no research studies which specifically investigate the enhanced reasoning mechanisms of LLMs for compiler optimizations.

VII. CONCLUSION

In this paper, we investigate whether conventional LLMs, such as Llama2, can perform basic peephole compiler optimization, aiming to understand their performance and the errors they produce. Our findings suggest that one of the reasons for these errors is absence of enhanced reasoning capabilities. To explore this, we tested most advanced AI models from, OpenAI GPT-4o and GPT-o1, that implement such capabilities. Our results show that GPT-o1, despite not being fine-tuned, significantly outperforms both fine-tuned Llama2 and GPT-4o. We attribute this improvement to the chain-of-thought reasoning implemented in GPT-o1. We hope our work will inspire further research on using LLMs with enhanced reasoning mechanisms and chain-of-thought for code generation and optimization.

REFERENCES

[1] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," 2023. [Online]. Available: https://arxiv.org/abs/2302.06590

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[3] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," 2020. [Online]. Available: https://arxiv.org/abs/2006.03511

[4] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[5] J. Armengol-Estapé and M. F. P. O'Boyle, "Learning c to x86 translation: An experiment in neural compilation," 2022. [Online]. Available: https://arxiv.org/abs/2108.07639

[6] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. P. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," 2024. [Online]. Available: https://arxiv.org/abs/2305.12520

[7] I. Hosseini and B. Dolan-Gavitt, "Beyond the c: Retargetable decompilation using neural machine translation," in *Proceedings 2022 Workshop on Binary Analysis Research*, ser. BAR 2022. Internet Society, 2022. [Online]. Available: http://dx.doi.org/10.14722/bar.2022.23009

[8] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Meta large language model compiler: Foundation models of compiler optimization," 2024. [Online]. Available: https://arxiv.org/abs/2407.02524

[9] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large language models for compiler optimization," 2023. [Online]. Available: https://arxiv.org/abs/2309.07062

[10] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[11] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Academic Press/Morgan Kaufmann, 1997.

[12] C. N. Fischer, R. K. Cytron, and R. J. L. Jr., *Crafting a Compiler*. Boston, MA, USA: Addison-Wesley, 2010.

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[14] LLVM, "Instructioncombining.cpp source file," Accessed: Jul. 7, 2024, Jul 2024, [Online]. Available: https://llvm.org/doxygen/InstructionCombining_8cpp_source.html.

[15] ——, "Aarch64mipeepholeopt.cpp source file," Accessed: Jul. 7, 2024, Jul 2024, [Online]. Available: https://llvm.org/doxygen/AArch64MIPeepholeOpt_8cpp_source.html.

[16] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open foundation and fine-tuned chat models," 2023. [Online]. Available: https://arxiv.org/abs/2307.09288

[17] OpenAI, "Gpt-4o system card," 2024. [Online]. Available: https://arxiv.org/abs/2410.21276

[18] ——, "O1 system card," 2024. [Online]. Available: https://cdn.openai.com/o1-system-card-20240917.pdf

[19] Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang, "Parameter-efficient fine-tuning for large models: A comprehensive survey," 2024. [Online]. Available: https://arxiv.org/abs/2403.14608

[20] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101

[21] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," 2023. [Online]. Available: https://arxiv.org/abs/2305.14314

[22] NVIDIA, "Cuda c programming guide," 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[23] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu, "Bleu: a method for automatic evaluation of machine translation," 10 2002.

[24] W. Xiaowei, W. Kuixing, and Y. Quansheng, "Research and development of compiler based on gcc," in *Recent Advances in Computer Science and Information Engineering*, Z. Qian, L. Cao, W. Su, T. Wang, and H. Yang, Eds. Berlin, Heidelberg: Springer, 2012, vol. 126, pp. 817–822.

[25] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024. [Online]. Available: https://arxiv.org/abs/2402.19173

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[27] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[28] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[29] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii,

T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[30] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," 2024. [Online]. Available: https://arxiv.org/abs/2312.02120

[31] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[32] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," 2021. [Online]. Available: https://arxiv.org/abs/2101.04808

[33] Z. Wang and M. O'Boyle, "Machine learning in compiler optimisation," 2018. [Online]. Available: https://arxiv.org/abs/1805.03441

[34] H. Leather and C. Cummins, "Machine learning in compilers: Past, present and future," *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–8, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221855481

[35] Y. Liang, K. Stone, A. Shameli, C. Cummins, M. Elhoushi, J. Guo, B. Steiner, X. Yang, P. Xie, H. Leather, and Y. Tian, "Learning compiler pass orders using coreset and normalized value prediction," 2023. [Online]. Available: https://arxiv.org/abs/2301.05104

[36] A. Haj-Ali, Q. J. Huang, W. S. Moses, J. Xiang, K. Asanovic, J. Wawrzynek, and I. Stoica, "Autophase: Juggling HLS phase orderings in random forests with deep reinforcement learning," in *Proceedings of the Third Conference on Machine Learning and Systems, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, I. S. Dhillon, D. S. Papailiopoulos, and V. Sze, Eds. mlsys.org, 2020. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2020/hash/5b47430e24a5a1f9fe21f0e8eb814131-Abstract.html

[37] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams, "Using machine learning to focus iterative optimization," 04 2006, pp. 11 pp.–.

[38] A. F. Zanella, A. F. da Silva, and F. M. Quintão, "Yacos: a complete infrastructure to the design and exploration of code optimization sequences," in *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*, ser. SBLP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 56–63. [Online]. Available: https://doi.org/10.1145/3427081.3427089

[39] H. Tan, Q. Luo, J. Li, and Y. Zhang, "Llm4decompile: Decompiling binary code with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2403.05286

[40] W. K. Wong, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Refining decompiled c code with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2310.06530

[41] X. She, Y. Zhao, and H. Wang, "Wadec: Decompiling webassembly using large language model," 2024. [Online]. Available: https://arxiv.org/abs/2406.11346

[42] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," 2023. [Online]. Available: https://arxiv.org/abs/2212.14834

[43] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language models," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, p. 709–735, Oct. 2024. [Online]. Available: http://dx.doi.org/10.1145/3689736

[44] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35. Curran Associates, Inc., 2022, pp. 24 824–24 837. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[45] C. Li, J. Liang, A. Zeng, X. Chen, K. Hausman, D. Sadigh, S. Levine, L. Fei-Fei, F. Xia, and B. Ichter, "Chain of code: Reasoning with a language model-augmented code emulator," 2024. [Online]. Available: https://arxiv.org/abs/2312.04474

[46] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang, "Cruxeval: A benchmark for code reasoning, understanding and execution," 2024. [Online]. Available: https://arxiv.org/abs/2401.03065

[47] J. Wang, M. Fang, Z. Wan, M. Wen, J. Zhu, A. Liu, Z. Gong, Y. Song, L. Chen, L. M. Ni, L. Yang, Y. Wen, and W. Zhang, "Openr: An open source framework for advanced reasoning with large language models," 2024. [Online]. Available: https://arxiv.org/abs/2410.09671

APPENDIX

Table VIII provides examples of typical optimizations performed by peephole.

TABLE VIII: Examples of typical optimizations performed by peephole.

Optimization Type	**Original Code**	**Optimized Code**	**Description**

Constant Folding	`mov w0, #2`
	`add w0, w0, #3`
	`ret`
		`mov w0, #5`
		`ret`
			The addition operation is folded into a single assignment.

Strength Reduction	`mov w1, w0`
	`mul w0, w1, #2`
	`ret`
		`lsl w0, w0, #1`
		`ret`
			Multiplication by 2 is reduced to a left shift operation.

Null Sequences	`lsl w8, w8, #1`
	`lsr w8, w8, #0`
	`ret`
		`lsl w0, w0, #1`
		`ret`
			Right shift by 0 is effectively a no-op.

Combine Operations	`lsl x2, x1, #1`
	`add x3, x2, x2`
		`add x3, x1, x1`
		`ret`
			Two expressions are combined into one.

Algebraic Laws	`mov w9, wzr`
	`mul w8, w8, w9`
	`ret`
		`mov w0, wzr`
		`ret`
			Any number multiplied by 0 is 0.

Address Mode Operations	`sub sp, sp, #16`
	`.cfi_def_cfa_offset 16`
	`mov w8, #1`
	`str w8, [sp, #12]`
	`ldr w0, [sp, #12]`
	`add sp, sp, #16`
	`ret`
		`mov w0, #1`
		`ret`
			Optimizing address mode operations.