Introducing Compiler Semantics into Large Language Models as Programming Language Translators: A Case Study of C to x86 Assembly

Shuoming Zhang[1, 3], Jiacheng Zhao[1, 3], Chunwei Xia[2],
Zheng Wang[2], Yunji Chen[1, 3], and Huimin Cui[*1, 3]

[1]SKLP, Institute of Computing Technology, CAS
{zhangshuoming21s,zhaojiacheng,cyj,cuihm}@ict.ac.cn
[2]University of Leeds, UK
{C.Xia, Z.Wang5}@leeds.ac.uk
[3]University of Chinese Academy of Sciences, Beijing, China

Abstract

Compilers are complex software containing millions of lines of code, taking years to develop. This paper investigates to what extent Large Language Models (LLMs) can replace hand-crafted compilers in translating high-level programming languages to machine instructions, using C to x86 assembly as a case study. We identify two challenges of using LLMs for code translation and introduce two novel data pre-processing techniques to address the challenges: numerical value conversion and training data resampling. While only using a 13B model, our approach achieves a behavioral accuracy of over 91%, outperforming the much larger GPT-4 Turbo model by over 50%. Our results are encouraging, showing that LLMs have the potential to transform how compilation tools are constructed.

1 Introduction

There is growing interest in using Large Language Models (LLMs) for software engineering tasks (Zhang et al., 2023b) like code retrieval (Li et al., 2022b,a), completion (Svyatkovskiy et al., 2020; Guo et al., 2023) and translation (Armengol-Estapé and O'Boyle, 2021; Armengol-Estapé et al., 2023). The training data of many LLMs, including CodeLlama (Rozière et al., 2022), Codex (Chen et al., 2021), and GPT4 (OpenAI et al., 2023) contains code examples. However, these models are not explicitly trained for code translation. Consequently, they are prone to errors during code translation (Armengol-Estapé et al., 2023). On the other hand, LLMs trained in natural language corpus have demonstrated impressive results in natural language understanding (Brown et al., 2020; Pruksachatkun et al., 2020). As such, it is interesting to know if LLMs can learn to compile code.

This paper investigates the feasibility of using Large Language Models to translate a high-level programming language to machine instructions, a problem known as *neural compilation* (Armengol-Estapé and O'Boyle, 2021). Traditionally, this is performed by a manually crafted compiler that usually takes many person-years of compiler engineers' time to build. Recent developments in LLMs have shown promising results in leveraging pre-trained Transformer models for tasks like decompilation (e.g., translating assembly code to C programs) (Armengol-Estapé et al., 2023) and program synthesis (Szafraniec et al., 2023). However, few works use LLMs as a compilation tool to translate a high-level programming language into low-level assembly instructions. We hypothesize that LLMs may be able to learn from examples generated from other means, e.g., code synthesis, but the learnt model can directly produce the translated or even optimized code. Our work seeks to bridge this gap by taking C to x86 assembly as a case study.

A key challenge we face is managing the semantic gap between high-level languages optimized for human usability and low-level languages designed for hardware executions. This gap often manifests in a lack of direct correspondence between elements of the source and target languages. For instance, some commonly used data structures and programming constructs in C, such as *struct* and complex *for-loop*, do not have single equivalent x86 instructions. Similarly, C uses identifiers for variables, while assembly instructions use stack and memory addresses or registers. As a single line of C code can be translated into a varying number of assembly instructions, learning the translation from C to assembly would require different amounts of training samples depending on the complexity of the mapping, making it difficult to construct a balanced training corpus.

To overcome the aforementioned challenges, we leverage Low-Rank Adaptation (LoRA) (Hu et al., 2021a) to fine-tune a pre-trained 13B CodeLlama

*Corresponding Author

model (Rozière et al., 2022). However, using the standard natural language training pipeline, our initial attempt yields a model with poor performance for C-to-assembly translations. After a close examination of the failure cases, we propose to introduce compiler semantics as two key data pre-processing techniques to enhance the trained model: symbolic interpretation for numerical value conversion and switch-case normalization for switch-case inconsistency. Furthermore, we propose an automatic compiler semantics guided refinement learning framework to improve the fine-tuned model iteratively. Our framework automatically resamples the distribution of semantic mapping samples and synthesizes the failure test cases in the validation set to improve the quality of the model training data.

We perform a large-scale evaluation on over 57k executable C programs and compare them against the state-of-the-art large language model GPT-4 Turbo. We verify the correctness of the generated x86 assembly code by executing them against unit test cases. Experimental results show that our neural compiler generates code that is more accurate than all competing baselines. Compared to GPT-4 Turbo, our approach improves the translation accuracy by over 50%, from 40.85% to 91.88%.

Our main contributions are:

- We propose an approach to introduce compiler semantics into the LLM as two new data preprocessing methods: symbolic interpretation and switch-case normalization. Experimental results demonstrate that the two proposed methods allow the LLM to increase the number of correct translations by over 30%.
- We implement an automatic refinement augmentation framework targeting the biased samples of different semantics in the corpora, where the long-tails under-fit. The framework resamples the semantics distribution by synthesizing incorrect cases, to obtain improved accuracy on the long tails.
- We can achieve 91.88% IO accuracy when translating C to x86 assembly and we believe it's the highest accuracy when comparing with SOTA works.

2 Problem Statement

We target the problem of machine translating high-level programs(specifically, in the C language) into semantically equivalent low-level programs(in x86 assembly) with limited bilingual parallel corpora. One way to generate the training data is to use an existing compiler, such as GCC, as an oracle to generate semantically aligned assembly code from C language corpora. However, there are other options like search-based code synthesis techniques (Hu et al., 2021b; Hu, 2022). Since training data generation is performed offline, the overhead of generating the corpora does not affect the end user of the LLM. Our approach is also useful in porting a pre-trained LLM to other hardware architecture, e.g., fine-tuning an LLM trained on C-x86 samples to generate ARM instructions from C programs. In this case, the pre-trained model can be fine-tuned on a small set of C-to-ARM-assembly samples generated through code synthesis, reducing the cost of targeting compilers for a new hardware architecture. Besides, Aligning different programming language other than C to x86 assembly is also possible, which we discuss in Appendix A.3.

Definition 1 *There is a high-level programming language $\mathcal{L}_{high}$ and a low-level programming language $\mathcal{L}_{low}$, each is an infinite set of valid program strings. There exists a unary relation $\rightharpoonup$ from $\mathcal{L}_{high}$ to $\mathcal{L}_{low}$. Given two monolingual corpora $L_{high} \subset \mathcal{L}_{high}$ and $L_{low} \subset \mathcal{L}_{low}$, the problem is to learn a translator F such that $\forall x \in \mathcal{L}_{high}, (\exists u \in \mathcal{L}_{low}, x \rightharpoonup u) \rightarrow (x \rightharpoonup F(x))$.*

Our main challenge in this work is the larger semantic gap between C and x86 assembly compared to translation between high-level codes like C-to-CUDA (Wen et al., 2022) and Java-to-Python (Rozière et al., 2020). For example, like *for-loop* and *if-else* semantics, the translation must learn **a posteriori** to generate jump instructions and corresponding labels to express the original control flows. According to (Rice, 1953), there is no set of rules that can accurately model the relation $\rightharpoonup$, because it is undecidable whether two programs are semantically-equivalent. Instead, we will use behavioral-equivalent to approximate.

3 Methodology

Our approach for translating high-level C code to low-level x86 assembly code targets generating semantically equivalent code in best efforts. In this work, we target non-optimized code generation, using the result given by the GCC compiler as a reference to train our model.

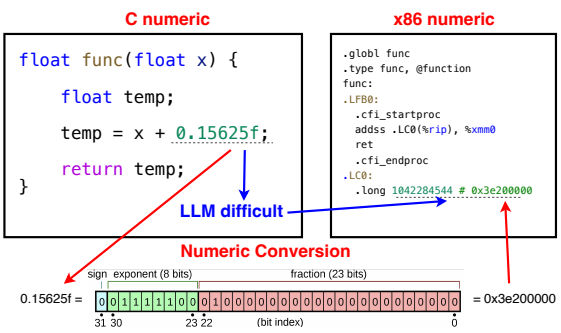

Figure 1: Numerical Conversion Feature Between C And x86

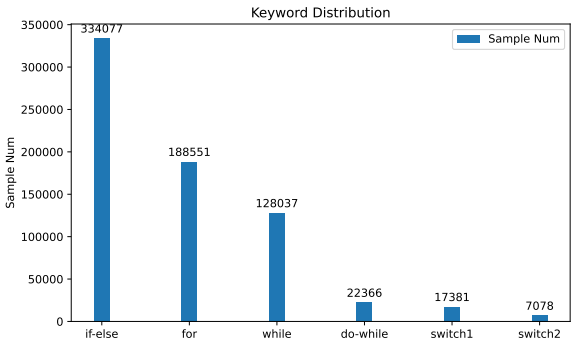

Figure 2: Long-tail Keyword Distribution of ExeBench

3.1 Dataset Preprocssing

To train a model, we first construct a C-x86-aligned bilingual corpora. We use benchmarks in the AnghaBench (Da Silva et al., 2021) and ExeBench (Armengol-Estapé et al., 2022) suites to obtain a large C corpora codebase. Then, we filtered C code with non-standard library dependencies and used GCC (version 9.4.0) with the "-O0" option to compile each program into x86 assembly.

After the initial preprocessing, we obtain a semantically aligned C-x86 bilingual corpora for training. However, a model trained directly on compiler-generated corpora does not perform well. After manually inspecting the generation errors, we find the following challenges.

Numerical Value Conversion. A significant challenge in the translation between C and x86 assembly languages lies in the conversion of numerical values, which underscores the semantic differences between these languages. As depicted in Figure 1, In C, floating-point and double-precision values can be represented as literals, such as 1.0 or 3e-5. However, in most compiler designs, these numerical literals need to be converted to an internal representation following the IEEE-754 standard (IEEE, 1985). This conversion process is rule-based and straightforward to implement. Yet, Large Language Models (LLMs) exhibit a notable weakness in this task, achieving a mere 3.8% accuracy on NumericBench, a large scale mathematical solving dataset derived from Math23K (Wang et al., 2017). This result underscores a critical limitation of LLMs in handling numerical computations.

To mitigate this limitation, we implement an effective data pre-processing method called symbolic interpretation, where we guide the LLM to generate symbolic expressions of the float/double values, which are subsequently processed by a rule-based interpreter. By delegating the actual numerical conversion to the interpreter, this method effectively circumvents the LLM's inherent weakness in numerical value conversions, thereby improving the overall accuracy of the translation process.

```c
int x;
int main() {
    switch(x) {
        case 0: ...
        case 1: ...
            ...
        case N: ...
        default: ...
    }
    return 0;
}
```

Listing 1: C Switch1

```c
int x;
int main() {
    {
        if(x == 0) ...
        else if(x==1) ...
        ...
        else if(x==N) ...
        else ...
    }
    return 0;
}
```

Listing 2: C Switch2

```
main:
    movl  x(%rip), %eax
    cmpl  $N, %eax
    ja  .Ldefault
    leaq  .LJT0(%rip), %rdx
    movslq  (%rdx,%rax,4), %rax
    addq  %rdx, %rax
    notrack jmp *%rax
.LJT0:
    .long   .L0-.LJT0
        ...
    .long   .LN-.LJT0

.L0:
    ...
...
.Ldefault:
    ...
```

Listing 3: x86 Switch1

```
main:
    movl  x(%rip), %eax
    cmpl  $N, %eax
    ja  .Ldefault
    cmpl  $0, %eax
    je      .L0
    cmpl $1, %eax
    je      .L1
        ...
    je      .LN
    jmp     .Ldefault
.L0:
    ...
.L1:
    ...
...
.Ldefault:
    ...
```

Listing 4: x86 Switch2

Switch-case Statement Inconsistency. Another kind of significant translation error lays on "switch-case" statement, where we observe that our baseline model generates inconsistently in two styles, where the compiler generated corpora messed them up. Listing 1 depicts the standard switch-case statement in C, and Listing 3 is its corresponding x86 assembly generated by GCC, where the cases are stored into a jump table, and using indirect jump instruction to control the jump target. However, switch-case statement can also be implemented by

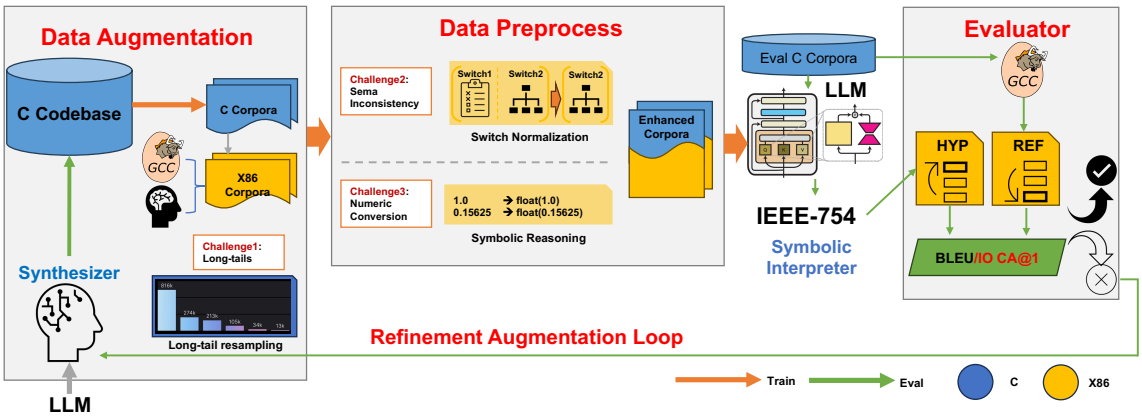

Figure 3: Data Augmentation Framework Overview

if-else logic, where Listing 2 depicts its semantic equivalent code in C, and Listing 4 is its x86 assembly, where multiple comparison instructions and conditional jump instructions are used. By default, GCC generates the first type when cases are larger than threshold 4, and the second type otherwise, other compilers like Clang and MSVC also sharing this behavior with different thresholds. As depicted in Figure 2, we observe 7078 samples belong to the first and 17381 samples belong to the second in our initial training corpora, and their ratio on the whole corpora is also small, with 1.0% and 2.6% respectively. Comparing to other control keyword in C, which is clearly long-tailed.

To tackle the switch-case semantic inconsistency, we normalize the semantic of the switch-case statement to the if-else style in Listing 4, where we re-generate the x86 assembly from GCC compiler using compiler flag "-fno-jump-tables".

3.2 Dataset Augmentation

As already emphasized in the switch-case handling, the biased distribution of each semantic translation in the training corpora is a big challenge. Considering there are other long-tails besides switch-cases that also performs poorly, we need an automatic data augmentation method to improve the model's accuracy on these long-tails. This is crucial and necessary because the LLM is only trained on limited corpora. If the input is few or even none in the corpora, it will translate poorly without any surprise.

Inspired by (Madaan et al., 2023), we construct an automatic refinement data augmentation framework as depicted in Figure 3, where the model is first trained on corpora from the previous method, and evaluated through multiple metrics, where we collect on the low-metric samples where we assume the model under-fits to learn them. Then we synthesize more samples from the incorrect samples to improve the distribution. we choose to use mistral-7B (Jiang et al., 2023a) as the synthesizing LLM in our implementation, where we instruct the LLM to analyze, categorize, and generate ten times more similar samples.

With more long-tail samples been synthesized, we re-sample the corpora by adding synthesized samples to it, creating a re-sampled corpora that better represents the long-tail problems. Finally, we re-train the model on this re-sampled dataset. The whole above process can be iteratively executed, where more under-fitting long-tails can be discovered, re-sampled, and improved.

This refinement framework allows the model to better learn how to handle these long-tailed samples, leading to improved accuracy in the generated low-level code. We provide examples illustrating its validity in the case studies.

3.3 Fine-Tuning

Machine translation has evolved significantly with the advent of neural machine translation (NMT), where models are trained on large corpora of text to learn the nuances of language translation. The general principle of machine translation, as pioneered by (Rozière et al., 2020), involves two key stages: pretraining and fine-tuning. Initially, models are pretrained on monolingual corpora to learn language features. Subsequently, they are fine-tuned on paired corpora to guide the translation between two languages.

We employ Low Rank Adaptation (Hu et al.,

Datasets	Size	Tok (C)	Tok (x86)
Train	679665	107	391
Train-Num	40000	168	594
Eval	57552	110	
ExeBench	35704	108	
Numeric	21104	111	
Switch	744	237	

Table 1: Dataset Details

2021a), one of the most well-known Parameter-Efficient Fine-Tuning methods, to adapt LLMs to our translation task. LoRA modifies a small subset of the model's weights by decomposing the weight changes into two smaller matrices, which are then fine-tuned. This approach allows us to bypass the initial pre-training phase typical in machine translation, as LLMs are already pretrained on extensive monolingual corpora. We use **codellama-13b** (Rozière et al., 2022) as our foundation model.

Similar to the construction of the training corpora, we construct the evaluation corpora solely on C, where we choose from the IO evaluation part of ExeBench (Armengol-Estapé et al., 2022) and Math23K (Wang et al., 2017), to evaluate the model's translation accuracy, where the former represents general purpose code and the latter represents numerical computations. More detailed corpora components can be found in the following Evaluation Section.

4 Evaluation

4.1 Dataset

To evaluate our proposed code translation methods, we perform a series of experiments on function-level C programs. We first finetune the codellama-13b foundation model to perform C-to-x86 code translation task, where we use dataset derived from ExeBench (Armengol-Estapé et al., 2022) and AnghaBench (Da Silva et al., 2021), two large scale dataset of compilable C functions, we first apply data cleaning, where we filtered oversized functions(we limit the size to 2048 tokens in our settings), and other features we are not going to cover like inline assembly. Finally we get a 680K size training dataset for baseline training. In the numerical value conversion preprocessing part, we establish a 40k numerical adjusted corpora to finetune the model. For evaluation part, we construct a 57K size dataset with I/O behavioral checks. As for the numerical conversion and switch-case generation challenges, we also categorize specified subsets in the evaluation, where a 21K numeric-specific subset and a 744 switch-specific subset are evaluated individually. Table 1 shows the details of the dataset we used in training and evaluation.

4.2 Setup and Metrics

We set up the experiment on a Ubuntu 22.04 server with Intel Xeon Platinum 8358 CPU and 4 x A800 80GB GPUs. We begin with the codellama-13b-instruct checkpoint from huggingface hub as our foundation model. We then directly apply LoRA finetuning with the 680K training corpora to learn the C-to-x86 translation task, which we considered as the **Baseline** model. Later we apply the two data pre-processing methods, switch-case normalization or/and numerical value conversion, to adjust the training corpora, and re-train on the foundation model to get the **Switch** enhanced model, **Numeric** enhanced model and **ALL** enhanced model. We also use more foundation LLMs as second baselines to compare with, where we majorly evaluate on **GPT**-4-Turbo Other foundation LLMs like GPT-4o, GPT-4o-mini, Llama-3.1-70B, Mixtral-8x7B, and code LLMs like DeepseekCoder, are also evaluated.

During the training process, we use **lora_r** = 128, **lora_alpha**=32, **lora_dropout**=0.05 in the LoRA modules, where we attach all **Q, K, V, O** in the model for training. We use the sum of token-level cross-entropy loss with teacher-forcing as the loss function, which is on par with (Rozière et al., 2020). We use AdamW(Kingma and Ba, 2014) as the optimizer and apply a cosine learning rate that top at 1e-4 in training. The training process is performed fully in float16 precision, where we train the model for 1 epoch in 70 hours using 4xA800 80GB GPUs.

We evaluate the above models on the 57,552 functions evaluation dataset. We also construct the 21,104 size numeric-specified and the 744 size switch-specified subsets from the full dataset. Then we perform end-to-end evaluation on these datasets, which also serves as an ablation study. We examine each generated function in x86 assembly by linking it with the driver code that called the function to obtain an executable, then performing Input/Output(IO) correctness checks. We use greedy generation in the generation process, so the IO accuracy can also be viewed as CA@1 or Pass@1 in other machine translation tasks.

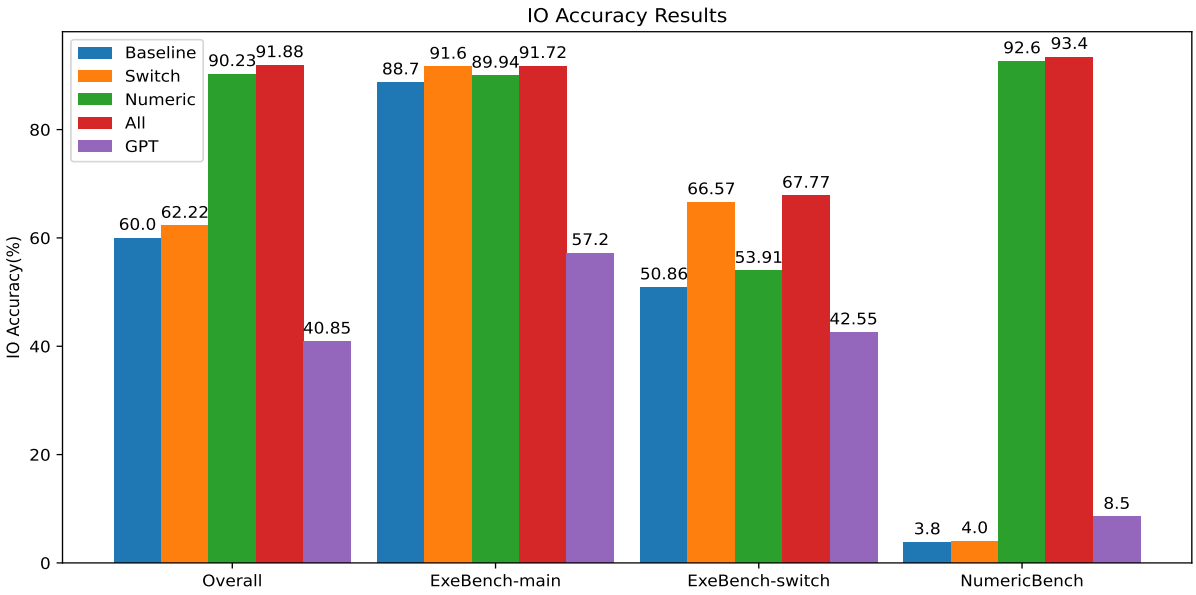

Figure 4: IO Accuracy Results

4.3 End-to-End Evaluation

Figure 4 summarizes the empirical end-to-end results ablating different methods and comparing with directly finetuned CodeLlama-13B as baseline and GPT-4-Turbo, the foundation LLM baseline. More LLM baselines can be found in Table 2. As the results suggest, the baseline model performs fairly well, achieving 60% overall accuracy and 88.7% in ExeBench, which outperforms all foundation LLM baselines. More detailed breakdowns of its wrong translations show it majorly falls into the following types:

Generating wrong numerical values. We capture all the functions within the evaluation dataset, where there exists numerical value initialization, and categorize them into a numerical dataset, NumericBench for breakdown. We find out that the baseline model can only generate 3.8% of NumericBench correctly, and most of these happen-to-be-correct values are values with high frequency in the dataset, like 1.0 and 0.0. This breakdown indeed reveals a crucial drawback of the LLM-based machine translation method. We then apply the symbolic interpretation method on the dataset preprocessing stage, which significantly improved the generation accuracy, rising from 3.8% to over 90%.

Generating wrong labels and jump tables. We evaluate the evaluation dataset and collect those with incorrect execution behaviors, where we find many in switch-case generations. After analyzing the generated assembly, we find out their translation is very likely in an underfitting manner. We also find out the training dataset is inconsistent with the semantic of switch-case code generation, when cases numbers are above the threshold, they use indirect jump on the jump table in the generated assembly, while the if-else style in the others. This inconsistent behaviour is by default open for our oracle compiler GCC even in O0 optimization level, where dataset makers can hardly notice.

We further perform categorization of control-flow statements on the training dataset, which is clearly summarized in Figure 2, where the two types of switch-case generation are both rare in corpora, counting for 2.6% and 1.0% respectively. This categorization result depicts a long-tail distribution in the training dataset, where the model under-fits the switch-case statement generation, and the inconsistency on switch-case statement generations may further confuse the model.

To tackle this problem, we perform switch-case normalization, where we enable the GCC option "-fno-jump-tables" to unify the generation behaviours on switch-case, and re-train the model. As illustrated in Figure 4, the normalization of switch-case semantic improves the switch-case translation accuracy from 50.86% to 66.57%, which shows the effectiveness of the augmentation method.

Other types of wrong generations, which include wrong generation of very long function logics, wrong generation of stack operation, wrong C-struct offset calculation, and wrong generation on rare samples, like AVX intrinsics, etc. More de-

Models	Accuracy
GPT-4-Turbo	57.2%
GPT-4o	68.9%
GPT-4o-mini	50.5%
DeepseekCoder	71.0%
Llama-3.1-70b	67.0%
Mixtral-8x7b	31.0%
Ours	**91.7%**

Table 2: More baseline models' accuracy comparison in ExeBench, all models are prompted to perform 0-shot neural compilation.

tailed evaluation results, like translation accuracy across code length can be found in Appendix B due to page limits.

In the end-to-end evaluation, we tackle the first two kinds of errors. By augmenting with both numerical conversion and switch-case normalization, we successfully improve the overall I/O Accuracy to 91.88%, which improves drastically from the baseline model. To compare with, GPT-4-Turbo can only achieve 40.85% I/O Accuracy even with careful prompting.

5 Case Study

We conduct case studies to demonstrate how to overcome the challenges using data augmentation methods to learn C-to-x86 translation.

The first case study demonstrate a function that need float/double numerical value conversion. In x86 language, float/double immediate numbers can not be encoded in instructions directly, and modern compilers like GCC save them in binary format following the IEEE-754 standard. So as long as the program exists numerical initialization, there are numerical conversions during the translation process, where LLMs perform poorly. As depicted in Figure 5, direct value conversion using implicit IEEE-754 rule makes LLMs hard to predict, where the baseline models are very likely to generate wrong numbers. By delegating the numerical conversions from LLMs to rule-based interpreters, where we augment the model to generate symbolic expressions instead of direct guessing, LLMs delegate the numerical conversion to rule-based interpreters, which can handle their conversions well, so that the numerical handling drawback of LLMs is efficiently mitigated.

The second case study depicted in Figure 6 shows the challenge of switch-case generation,

```
float func()
{
    float costA = 6.0;
    float costB = 0.125;
    float cash = 50.0;
    float numA = 4.0;
    float numB;
    float temp;
    temp = costA * numA;
    temp = cash - temp;
    numB = temp / costB;
    return numB;
}
```

```
func:
  ...
  movss .LC0(%rip), %xmm0
  movss %xmm0, -20(%rbp)
  ...
.LC0:
  .long 1086324736  ; 6.0
.LC1:
  .long 1040187392  ; 0.125
.LC2:
  .long 1112014848  ; 50.0
.LC3:
  .long 1082130432  ; 4.0
```

Figure 5: Case Study 1: Numerical Conversion

```
int color_char_to_attr(char c)
{
    switch (c)
    {
        ...
    case 'R':
        return (4);
    case 'G':
        return (5);
    case 'B':
        return (6);
        ...
    }
    return (-1);
}
```

```
color_char_to_attr:
  ...
  subl  $66, %eax  ; 'B'
  cmpl  $16, %eax  ; 'R' - 'B'
  ja  .L2
  leaq  .L4(%rip), %rdx
  movslq  (%rdx,%rdi,4), %rax
  addq  %rdx, %rax
  notrack jmp *%rax
  .section .rodata
.L4:
  .long .L2-.L4
  .long .L2-.L4
  .long .L2-.L4
  ; ... repeated pattern
```

Figure 6: Case Study 2: Switch Generation

where the jump-table style generation are hard to learn for LLMs. The baseline model fails in the generation of jump table items, causing repeated patterns until the maximum generation length. By leveraging the if-else style data augmentation, the model has learned to treat switch-case statements as if-else style, where if-else corpus are on the head of keyword distribution with hundreds of thousand samples comparing to the rare long-tails, the deficient learning of switch-case generation is also mitigated.

The last case study shows how our refinement framework improving the long-tails performance. As depicted in Figure 7, AVX instructions are the SIMD extension in x86 assembly language, and is encapsulated as AVX intrinsics to be used in C language.

Recalling Figure 3, we introduce the refinement framework to augment the incorrect generations, which is inspired by (Madaan et al., 2023). Initially, there are no AVX-related samples in the training corpora at all, where the model without any surprise translate incorrectly without apriori. Then the incorrect AVX sample is captured by the evaluator together with other incorrect samples. we then use LLM to analyze the C code, and synthesize more based on several rules as prompts to generate more C samples closely related to the incorrect cases.

```
void foo(float *x, *y, *o) {
    xx = _mm256_load_ps(x);
    yy = _mm256_load_ps(y);
    zz = _mm256_add_ps(xx, yy);
    _mm256_store_ps(o, zz);
}
```

```
foo:
    vmovaps (%rdi), %ymm0
    vmovaps (%rsi), %ymm1
    vaddps  %ymm1, %ymm0
    vmovaps %ymm0, (%rdx)
    ret
```

Figure 7: Case Study 3: AVX Intrinsics Learning

We use mistral-7B(Jiang et al., 2023a) as the synthesizer LLM in our implementation. Finally, the sythesized augmented C corpora of incorrect samples is added back to the training dataset, where retraining/finetuning can be performed depending on the need.

Back to the case itself, a 10x synthesizing is sufficient enough to learn a new feature with simple semantic pattern, like the $\_mm256\_add\_ps$ intrinsic in the case, which simply generates a $vaddps$ instruction. Such LLM's learning ability of aligning C and x86 semantics is very impressive, which shows the few-shot learning potential in the language translation task. Although more complex patterns need more cases to learn well, luckily, the refinement framework can be executed iteratively, which can resample the corpora based on the generation accuracy, so that more complex cases can get more samples to be learned.

6 Related Work

Code Translation aims to translate a piece of code (usually a function or method) into another programming language. Early studies like (Nguyen et al., 2015) uses traditional statistical machine translation method. Neural-based method like (Chen et al., 2018) starts to be dominant, and capture the tree structure of programming languages. The emergence of pre-trained language models of code, such as CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021), has further improved the state of code translation. Large Language Models(LLMs) (OpenAI et al., 2023; Rozière et al., 2022) have continued this trend, showing promise in code translation task. However, the above approaches usually require fine-tuning on parallel corpora, which is often scarce.

Data augmentation techniques have been extensively used and found effective in machine translation tasks, which served as a solution to the scarcity of parallel corpora. Transcoder (Rozière et al., 2020) first propose back translation approach to learn unsupervised code translation, where the back-translation process also generates an automatic parallel corpora augmentation method. Transcoder-ST (Roziere et al., 2021), CodeXGlue (Lu et al., 2021), BabelTower (Wen et al., 2022) and CMTrans (Xie et al., 2023) also follow this approach, to obtain parallel corpora during the learning process. Besides direct generation, (Szafraniec et al., 2023) explores an IR-in-the-middle approach, while (Tang et al., 2023; Ahmad et al., 2023) both introduce an intermediate code summary stage, to improve the code translation accuracy.

To construct a balanced corpora in limited size in monolingual language is also challenging, it is naturally in a long-tailed distribution for different aspects of code semantics. where neural models tend to perform low accuracy on the tails due to lack of samples. (Zhout et al., 2023) reveals that LLMs can perform between 30% to 254% worse in long-tailed cases, where the model under-fits them. Inspired by the survey of long-tailed learning (Zhang et al., 2023a), we establish a refinement augmentation method, where long-tailed C samples are recognized in the evaluation process via metrics, then analyzed, synthesized by another powerful LLM, compiled by GCC to obtain parallel samples, finally augmented the corpora with more long-tailed knowledge.

Cross Level Code Translation. On high-level code to low-level code translation researches, (Armengol-Estapé and O'Boyle, 2021) first gives a try of using neural machine translation on this scenario. (Guo and Moses, 2022) further studies on C-to-LLVM IR translation. However, they only perform limited investigations on the methods, and their results are still on the preliminary stage. There are more related works on the reverse process, to recover high-level code from low-level code (Fu et al., 2019; Cao et al., 2022; Armengol-Estapé et al., 2023).Unlike the difficulty on semantic mapping to low level code in our challenges, their challenges mainly are on optimization recovery and type inference, while the semantic recovery is relatively simpler.

7 Conclusion

Machine translation from high-level language to low-level machine instructions is difficult. Even using advanced LLMs can not reach high accuracy directly. By implementing symbolic interpretation and switch-case normalization, two novel data preprocessing methods, we overcome numerical value

conversion and switch-case semantic inconsistency, two significant challenges in C-to-x86 language translation.

To improve the accuracy on long-tailed samples where the model under-fits to learn, we propose an automatic refinement augmentation framework to obtain improved accuracy on the long-tails by using synthesizing method on incorrect cases.

Finally we achieve state-of-the-art IO accuracy, over 91%, when translating C-to-x86 on a large-scale evaluation dataset. Comparing to LLM-only method(GPT-4-Turbo, 40.85%), and finetuning-only baseline method(59.87%), the methods show great efficiency. More importantly, we show LLMs can perform well in C-to-x86 neural compilation task, and potentially other language pairs.

In conclusion, these advancements demonstrate the potential of combining compiler-semantic-guided data pre-processing and augmentation techniques with LLMs to significantly enhance machine translation accuracy, paving the way for future innovations in tasks like neural compilation.

8 Limitations

We identify three main limitations in our work.

First, We currently use LoRA finetuning on open-weighted LLMs as our learning method instead of full-training due to resource constraints. We currently only research on C-to-x86, one of the most representative machine translation tasks across semantic levels. But the ideas of automatically augmenting the dataset with more balanced distribution, offloading numerical conversions from LLMs and unifying necessary semantics in the corpora are also applicable to other similar translation tasks. We investigate the generality of our methods on different assembly languages in Appendix A.1, which shows that the methods proposed in this work are applicable to a large scope of assembly languages in modern architectures.

Second, introducing code optimization is another research topic in code translation, where the model not only translates the source code to target code, but also performs optimizations. We don't target optimizations because the translation problem is not studied well yet. Like the numerical conversion problem in our unoptimized translation settings, there will be more similar problems that LLMs need to adjust to. We consider this as future work.

Third, our model learns the translation process by performing supervised fine tuning on foundation model, so there will be need for aligned C-x86 code corpora, which can be generated in multiple ways. We use an oracle compiler to generate code pairs for training in this work, as for other possible ways to translate on other language to other assembly, we discuss it in Appendix A.3.

9 Ethical Impact

In this work, we majorly study the effectiveness of using supervised fine-tuning on LLMs to translate C language to x86 language, where the researching subject is highly overlapped with compilers. However, this work doesn't seek to replace compilers but as an assistant for agile compiler developments.

As for the machine translation community, this work is to our knowledge the first to study the empirical effort on how to translate a high level programming language to assembly well, and what are the challenges.

We don't find any clear ethical problems during our research. All datasets and models we use in this work is publicly available. Although unlikely but possibly, the model fine-tuned for assembly code may contain vulnerability for execution. However, techniques like sandbox isolation (Wu et al., 2024) can be helpful to mitigate such concerns where the code is executed in an isolated environment.

10 Societal Impact

In this work, by finetuning LLMs as neural code translators in C-to-x86 compilation and achieving over 91% behavioral accuracy, we validate that LLMs can be potentially used for machine translation tasks from a high-level language to a low-level language. We expect more findings to the research problem of neural compilation. For example, problems like scalability, optimization and linguistical breakdowns in neural compilation, can be seen as future work.

Acknowledgements

This work is partially supported by National Natural Science Foundation of China (U23B2020, 62232015, 62302479) and Innovation Project E361010 of ICT, CAS. We would like to thank Zhicheng Li, Zhongcheng Zhang, Lei Qiu and Professor Xiaobing Feng from SKLP, ICT, CAS as well as Guanyu Qu and Yu Xu for discussions and proofreading throughout this research.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2023. Summarize and generate to back-translate: Unsupervised translation of programming languages. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pages 1520–1534. Association for Computational Linguistics.

Jordi Armengol-Estapé and Michael FP O'Boyle. 2021. Learning c to x86 translation: An experiment in neural compilation. *arXiv preprint arXiv:2108.07639*.

Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael FP O'Boyle. 2022. Exebench: an ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 50–59.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O'Boyle. 2023. Slade: A portable small language model decompiler for optimized assembler. *arXiv preprint arXiv:2305.12520*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Mihai Budiu and Chris Dodd. 2017. The p416 programming language. *ACM SIGOPS Operating Systems Review*, 51(1):5–14.

Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. 2022. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 508–518.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA. Curran Associates Inc.

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

Anderson Faustino Da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. 2021. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 378–390. IEEE.

Maxim Enis and Mark Hopkins. 2024. From llm to nmt: Advancing low-resource machine translation with claude. *arXiv preprint arXiv:2404.13813*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian Mcauley. 2023. LongCoder: A long-range pre-trained language model for code completion. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 12098–12107. PMLR.

Zifan Carl Guo and William S. Moses. 2022. Enabling transformers to understand low-level programs. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021a. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jingmei Hu. 2022. *Improving Assembly Synthesis via Interaction and Parallelism*. Ph.D. thesis, Harvard University.

Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L Glassman. 2021b. Assuage: Assembly synthesis using a guided exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, pages 134–148.

IEEE. 1985. Ieee standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023a. Mistral 7b.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2023b. Nova$^+$: Generative language models for binaries.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang, and Yanlin Wang. 2022a. Exploring representation-level augmentation for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen, and Nan Duan. 2022b. Soft-labeled contrastive pre-training for function-level code representation. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 118–129, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 585–596.

OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mo Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David

Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2023. Gpt-4 technical report.

Yada Pruksachatkun, Jason Phang, Haokun Liu, Phu Mon Htut, Xiaoyi Zhang, Richard Yuanzhe Pang, Clara Vania, Katharina Kann, and Samuel R. Bowman. 2020. Intermediate-task transfer learning with pretrained language models: When and why does it work? In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5231–5247, Online. Association for Computational Linguistics.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2022. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773*.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA. Association for Computing Machinery.

Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. 2023. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Marc Szafraniec, Baptiste Roziere, Hugh James Leather, Patrick Labatut, Francois Charton, and Gabriel Synnaeve. 2022. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations*.

Zilu Tang, Mayank Agarwal, Alexander Shypula, Bailin Wang, Derry Wijaya, Jie Chen, and Yoon Kim. 2023. Explain-then-translate: an analysis on improving program translation with self-generated explanations. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 1741–1788, Singapore. Association for Computational Linguistics.

Guido Van Rossum et al. 2007. Python programming language. In *USENIX annual technical conference*, volume 41, pages 1–36. Santa Clara, CA.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Yan Wang, Xiaojiang Liu, and Shuming Shi. 2017. Deep neural solver for math word problems. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 845–854, Copenhagen, Denmark. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

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

Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yongwei Zhao, Xing Hu, Zidong Du, Ling Li, et al. 2022. Babeltower: Learning to auto-parallelized program translation. In *International Conference on Machine Learning*, pages 23685–23700. PMLR.

Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. Refining decompiled c code with large language models. *arXiv preprint arXiv:2310.06530*.

Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. 2024. Secgpt: An execution isolation architecture for llm-based systems. *arXiv preprint arXiv:2403.04960*.

Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. 2023. Data augmentation for code translation with comparable corpora and multiple references. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13725–13739, Singapore. Association for Computational Linguistics.

Yifan Zhang, Bingyi Kang, Bryan Hooi, Shuicheng Yan, and Jiashi Feng. 2023a. Deep long-tailed learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023b. A survey on language models for code. *CoRR*, abs/2311.07989.

Xin Zhout, Kisub Kim, Bowen Xu, Jiakun Liu, DongGyun Han, and David Lo. 2023. The devil is in the tails: How long-tailed code distributions impact large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 40–52. IEEE.

A Discussions

Due to page limits, we discuss some interesting problems and challenges we find in our work as follows.

A.1 Method Availability

We majorly discuss the C-to-x86 machine translation in this paper, and developed symbolic interpretation method on floating value conversion problem and switch-case normalization on x86 assembly. People may have concerns about the generality of the methods we proposed. In fact, we analyzed and studied the usability of the symbolic interpretation and switch-case normalization method on three other trending architectures, including ARM, MIPS and RISCV. We will explain the availability of our methods by answering the following research questions(**RQs**).

RQ1: ***How does numerical values stored in each architecture? What are the challenges?*** In most cases, floating-point values are stored through IEEE-754 standard (IEEE, 1985), and thus these values are facing the either explicit or implicit conversion during the compilation process. Integer values, although not facing the conversion challenge, each architecture has different instruction bit preserved to use immediate values, for example, MIPS architecture only allows a 12-bit integer offset in its arithmetic instructions. Values that over 12-bit field should be stored in memory, and use load instruction to load them.

As far as we studied in x86, immediate values overflow is very rare in our test cases, because most immediate values are capable of fitting in the instruction format in x86, since x86 is CISC and allows multiple instruction format.

However, when studying RISC-like architectures, like ARM, MIPS and RISCV, there will be new problems in immediate values usage. We have discovered a few. For example, integer values need to be guarded whether they can fit in RISC instruction format. Because these knowledge are implicit in text during training and the model is very hard to learn through compiler generated pairs, where some of them use immediate values directly in the arithmetic instructions(can fit) and some choose to load in memory first(can not fit).

As the answer to **RQ1**: *We identify that the floating-point conversion is necessary in these architectures just like x86. Besides, we even find more challenges in the immediate integer value usage in RISC-like architectures.* Since our major research language is x86 assembly in this work, we report them to the community and plan to solve them as future work.

RQ2: ***How can numerical conversion be applied to these architectures?***

We will only use float values for illustration since double values are similar. In x86 architecture, the floating values are stored in IEEE-754 converted integer values in memory by compilers, which is hard for the model to learn the conversion rule currently. We choose to offload the conversion to a rule-based converter so that the model can leave the value conversion and keep the original float values as output. To achieve this, we also need to preprocess on the training dataset to teach the model to generate a format that will be recognized by the rule-based converter, and do not perform conversion.

When the target language is not x86, we need to study the availability of the numerical conversion method itself. Luckily, for numerical values, both ARM, MIPS and RISCV use similar format in the assembly language, just like x86. We use the float value 1.0f as an example. Listing 5 shows how float x is stored in memory and being loaded in x86. Similarly, Listing 6, Listing 7 and Listing 8 is the corresponding format in MIPS, ARM and RISCV respectively.

In the instruction part, all these assemblies just use symbol **x**, and the value itself is stored in the data section where x is assigned with value 1.0(0x3f800000). Since the pattern is almost identical to x86, we can surely apply the numerical conversion method on floating values in these architectures.

```
    ...
    movss x, %xmm0
    ...
x:
    .long 0x3f800000
```

Listing 5: x86 float

```
    lui $1, %hi(x)
    lwc1  $f2, %lo(x)($1)
    ...
x:
    .4byte 0x3f800000
```

Listing 6: MIPS float

```
    ...
    adrp  x8, _x@PAGE
    ldr s1, [x8, _x@PAGEOFF]
    ...
_x:
    .long 0x3f800000
```

Listing 7: ARM float

```
    ...
    lui a1, %hi(x)
    lw  a1, %lo(x)(a1)
    ...
x:
    .word 0x3f800000
```

Listing 8: RISCV float

RQ3: ***How can switch-case normalization be applied to these architectures?***

These architectures all support use either jump table style or if-else style to implement switch-case statement in C, just like x86 architecture. So the problem is also back to compiler implementations. Both GCC and Clang generate jump table style assembly when the cases are many and generate if-else style when small in all these architectures, even in -O0 optimization level. So it is difficult for the model to learn the implicit generation rules within limited switch-case corpora. Fortunately, we can use "-fno-jump-tables" option to align the compiler behaviors despite of case numbers. So the normalization method is applicable to these architectures.

In general, the problems of storing numerical values and aligning switch case statements in assembly languages are similar to x86 language we studied. Although some languages may not use switch statements, for example, **Lua** (Ierusalimschy et al., 2007) and **Python**(version <3.10) (Van Rossum et al., 2007), and some architectures like **P4** (Budiu and Dodd, 2017) may only use integer values, where our proposing methods are not applicable. However, the machine translation problem is also simpler and straightforward to implement, so as our answer to **RQ2** and **RQ3**: *Both numerical conversion and switch-case normalization methods can be applied to multiple different architectures.*

A.2 Impact on Large Language Model evolution

Our method uses supervised fine-tuning(SFT) on LLM with parallel code corpora in C and x86 to learn the machine translation process. However, neither SFT nor LLM is necessary for the machine translation task. We categorize current approaches into the following three categories.

- **Language Modeling + SFT:** Works like (Rozière et al., 2020) and (Szafraniec et al., 2022) use this methodology. Majorly they use smaller models like transformers (Vaswani et al., 2017). They first learn the model on monolingual corpora through language modeling, so that the base model learns the syntax and semantic of each language itself. Then they perform supervised fine-tuning(SFT) on language pairs, where the translation rules are learned. This is a natural thought on code translation and is the mainstream approach.

- **Pretrained Model + SFT:** As generative AI entered Large Language Model era, the foundation model itself learns huge amount of code corpora, which frees the need to perform language modeling on monolingual corpora, and developers can directly perform supervised fine-tuning on the foundation model, to teach the model about the translation process. Our work belongs to this category, as well as other works like (Wong et al., 2023; Cummins et al., 2023; Jiang et al., 2023b).

- **Pretrained Model only:** For the most advanced LLMs (OpenAI et al., 2023; Enis and Hopkins, 2024), which are trained on enormous amount of code in each language, including assembly language like x86 and ARM. They already learn the syntax and semantic of each language, so they can also perform machine translation on these languages directly. During our evaluation using GPT4(gpt-4-0613), although the performance is worse

than supervised fine-tuned models, their performance is still impressive, and some GPT4 generated translation has clearly learned the compilation process. There are potentials to use fine-tuning free methods, like Chain-of-Thought (Wei et al., 2022) and Retrieval-augmented generation (Lewis et al., 2020), to achieve better performance on code translation and gain great flexibility than SFT methods.

As Large Language Models keep evolving, we can look forward to more empirical methods on code translations, where utilizing the LLM on its understanding on code, can not only perform code translation tasks, but more complex ones like code optimizations, automatic bug solving, etc. Back to the neural compilation task itself, stronger LLMs may be able to generate more reasonable translation, and even generalize to translations on either new programming language features or new architecture features, which are truly helpful to compiler development and programming language designs.

A.3 Dependency on existing compilers

RQ4: ***Without an available compiler from one language to another, is the method still available?***

To answer this question, we first revise on where the compilers are used in our work. We use compilers as oracle to generate semantically aligned C-x86 corpora, where C-x86 compilers are powerful and near 100% correct. However, to obtain another programming language-x86 corpora, we don't necessarily need an existing compiler. We provide two possibllities.

A.3.1 Bridging to other code translation

This work majorly study on C-x86 code translation, a typical neural compilation task. In order to replicate our approach for other language pairs, there will be more challenges, the most important one is: bilingual neural-compilation corpora for some languages can be scarce, especially for non-compiled langauges like Python. Below is our discussions:

First, if there is a compiler between two languages, we can always obtain the parallel corpora for these two languages. If not, we can seek to find an intermediate language for bridging other code translations. Plenty of work on code translation (Rozière et al., 2020; Wen et al., 2022; Szafraniec et al., 2023) already provided methods to align high level programming languages' semantics, which

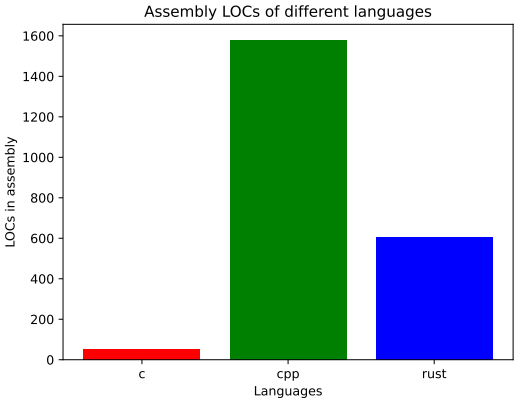

Figure 8: Assembly LOCs of different languages

makes them capable of generating a behavioral equivalent bilingual corpora between high level languages, for example, Python and C. By using an intermediate language, performing neural code translation first, our supervised fine-tuning(SFT) method is also applicable to learn Python-x86 compilation by compiling the C code to x86 assembly.

We also examine the choice on the intermediate language, because we can use other compiled language compilers to generate assembly as well. In comparison, Java, Python and JavaScript are interpreted languages that not suitable for compilation scenario. Other compiled languages, like C++ and Rust, are introducing more complex features like name mangling, implicit function execution and heavy standard library code injection, which is causing the generated assembly more complex and difficult to be learned.

For example, name mangling is initially a technique in compiler implementation to avoid symbol name conflict, however, its mangling rule is not easy for LLM to learn. However, we think it is solvable by similar techniques like symbolic interpretation for numerical values in this paper.

Other features, however, are more difficult to treat. As depicted in Figure 8, semantically equivalent C++ and Rust programs generate much longer assembly code compared to C, and many features are implicit functions like constructors and templates. These features are syntax sugar to programmers, but they are hard for either compiler implementation or neural model learning.

To compare with, C language has a minimal standard library, no name mangling mechanism and is explicit in its function execution, which makes C more friendly to be used as both the studied

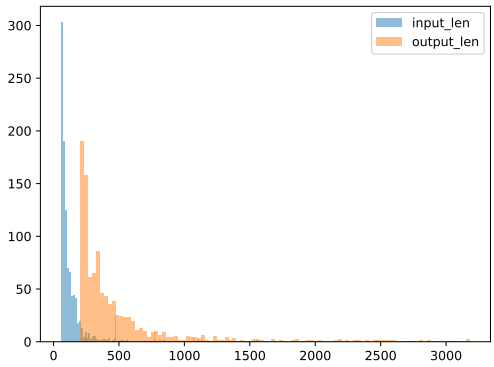

Figure 9: Model's I/O Length Distribution in ExeBench

language and the intermediate to connect assembly with other high-level languages.

A.3.2 Do we really need supervised fine-tuning?

Besides, with LLMs becoming more powerful, the need on using SFT for code translation is also questionable. If a LLM understands every line of code in each language, it will be able to align code in different languages, even one language is high level C and the other is low level x86 assembly.

Our evaluation results on GPT4 (gpt-4-0613) is quite impressive to reach over 40% accuracy in one-shot learning. We believe that with more powerful LLM and more prompting techniques (Wei et al., 2022; Lewis et al., 2020), there are more potentials for LLMs in code translation, especially neural compilation.

B Evaluation Details

RQ5: ***How does the model perform on long or short cases?***

During the evaluation process, we evaluate the model's behavioral accuracy through IO test. However, around **37%** of the IO cases provided by ExeBench(Armengol-Estapé et al., 2022) is not GCC-executable. We analyze the reasons, and some are caused by non-standard library usage, which is fixable, while some are wrong cases in its code patterns, which is hard to fix. In the end, we filtered these GCC-not-compilable cases.

We also analyze the statistics of translating on ExeBench. The distribution on Input/Output length is as depicted in Figure 9, where the generated x86 assembly length is about **3.56x** more than the input C length. The average C length is 108 token and the generated x86 length is 384 token.

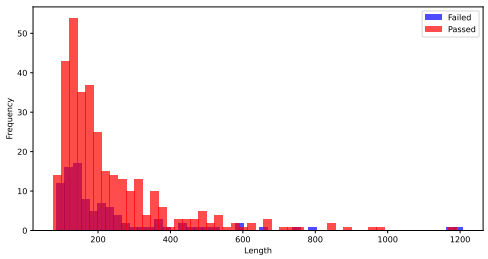

Figure 10: Input Length Distribution in Switch

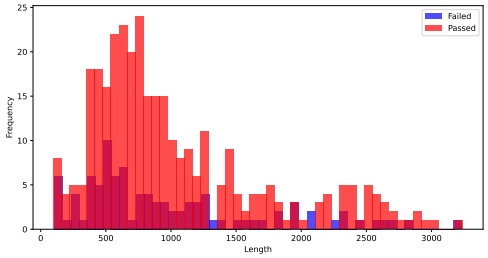

Figure 11: Output Length Distribution in Switch

Further analysis on the generation result shows that LLM tends to generate higher accuracy when the code size is small, and lower when code is large. This is natural since LLM is probabilistic and as the code size increases, the more likely errors may occur. The Switch case subset in ExeBench has 237 token size in C and 1032 token size in x86, which is double larger than the average ExeBench, while its generation accuracy is also lower, only 67.7% comparing to 91.72% in ExeBench.

As depicted in Figure 10 and Figure 11, Long input are more likely to fail in translation comparing to short input. However, the accuracy for very long input is still considerable, as the passed cases almost cover the failed cases in Figure 10, even for cases where the code size exceeds 1000 token.

RQ6: ***Why use 2048 as the context length and filtering size for finetuning?***

In comparison, many existing code translation work limits their code size to much smaller values like 128 or 512, either because of the model's capability or the training cost. In our settings, we choose the context size to 2048, which is significantly larger than previous work. 2048 is a tradeoff size for us to finetune on codellama-13b with 4xA800 80GB using a 64 batch size in total, which balanced training cost and performance. Theoretically we can increase the context size as long as it doesn't exceed the size of the foundation model(16384).