The Effectiveness of Transformer for Analyzing Low-Level Languages

Zifan (Carl) Guo - St. Mark's School

Mentor: William S. Moses

MIT PRIMES Conference - Oct. 18th

Compiler


#include<stdio.h>
int main()
{
printf("Hello, World!\n");
return 0;
}
hello_world.c
Compiler
0110011000100010011000111
1100000001111111110000001
1111000110101010001100011
0011000100010011000111110
0000001111111110000001111
1000110101010001100011001
1000100010011000111110000
0001111111110000001111100
0110101010001100011001100
0100010011000111110000000
1111111110000001111100011
hello_world.o

Series of Transformation

- High Level Languages
- (C, Java, Python)
  - High abstraction
  - English-like

- LLVM-IR (Intermediate Representation)
  - Less abstract but still readable
  - Platform independent

- Assembly Language
  - Even less abstract and readable
  - Platform dependent
    - X86_64
    - AArch64
    - RISC-V

- Machine Language
  - Not readable
  - 1s and 0s

Example: sum

C

```c
int sum(int num1, int num2){
    return num1 + num2;
}
```

LLVM-IR

```llvm
define dso_local i32 @sum(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}
```

X86-64 Intel:

```asm
sum:
        .cfi_startproc
# %bb.0:
        push    rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov     rbp, rsp
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], edi
        mov     dword ptr [rbp - 8], esi
        mov     eax, dword ptr [rbp - 4]
        add     eax, dword ptr [rbp - 8]
        pop     rbp
        .cfi_def_cfa rsp, 8
        ret
.Lfunc_end1:
        .size   sum, .Lfunc_end1-sum
        .cfi_endproc
```

Compiler Optimization

- Code transformation to make the program run faster (under the hood)

```
__attributes__((const))
double mag(int n, const double *A){
    double sum = 0;
    for(int i = 0; i < n; i++){
        sum += A[i] * A[i]
    }
}
void norm(int n, double *restrict out,
        const double *restrict in){
    for(int i = 0; i < n; i++){
        out[i] = in[i] / mag(n, in);
    }
}
```

```
void norm(int n, double *restrict out,
            const double *restrict in){
    double precomputed = mag(n, in);
    for(int i = 0; i < n; i++){
        out[i] = in[i] / precomputed;
    }
}
```

Unoptimized: $\Theta(n^2)$ → Loop invariant code motion (LICM) → Optimized: $\Theta(n)$

Compiler Optimization cont.

- Compilers provide standardly named optimization flags, such as -O1, -O2, -O3, or -Os
  - Never the best but good enough for casual users
- Need to figure out what optimization combinations are best:
  - Two biggest issue: optimization selection and phase ordering
  - Previously compilers adopt rule-based analyzers that are resource-intensive and error-prone
  - Now a growing trend to use machine learning models to replace rule-based ones

Transformers

- Attention is all you need (Vaswani et al. 2017)
- Unlike RNN or LSTM, not sequential → no locality bias
  - No drop in performance for long-distance context
- Allow parallel computation that saves time
  - Process sequences as a whole instead of word by word

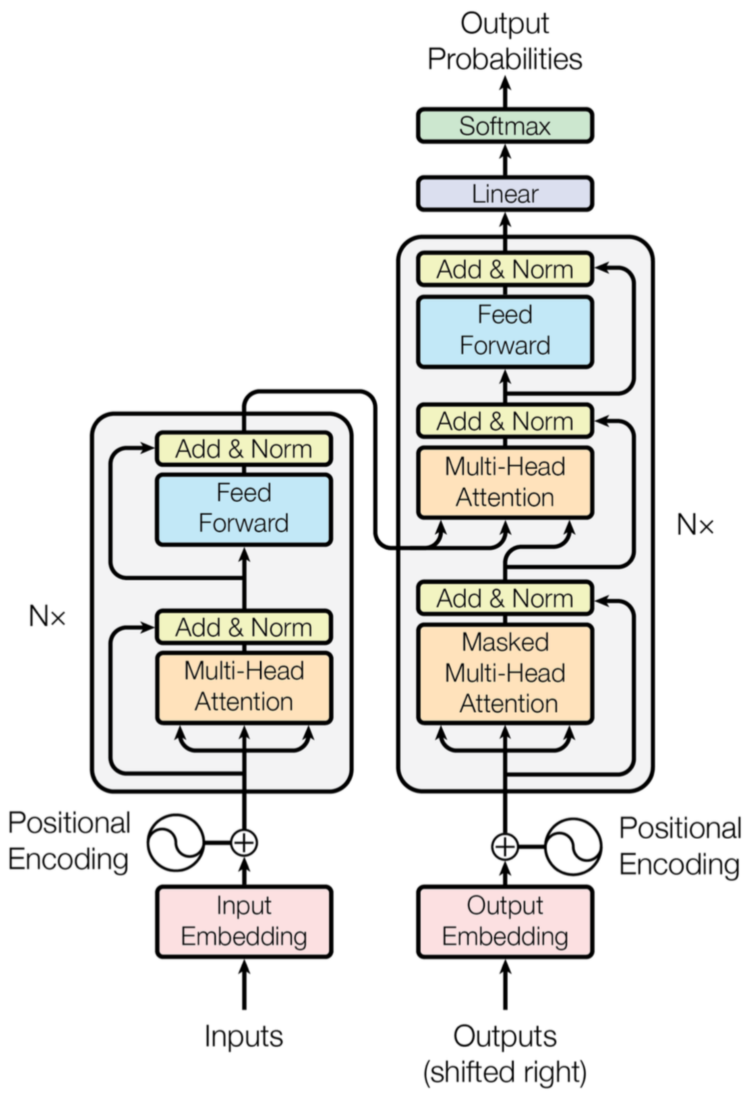
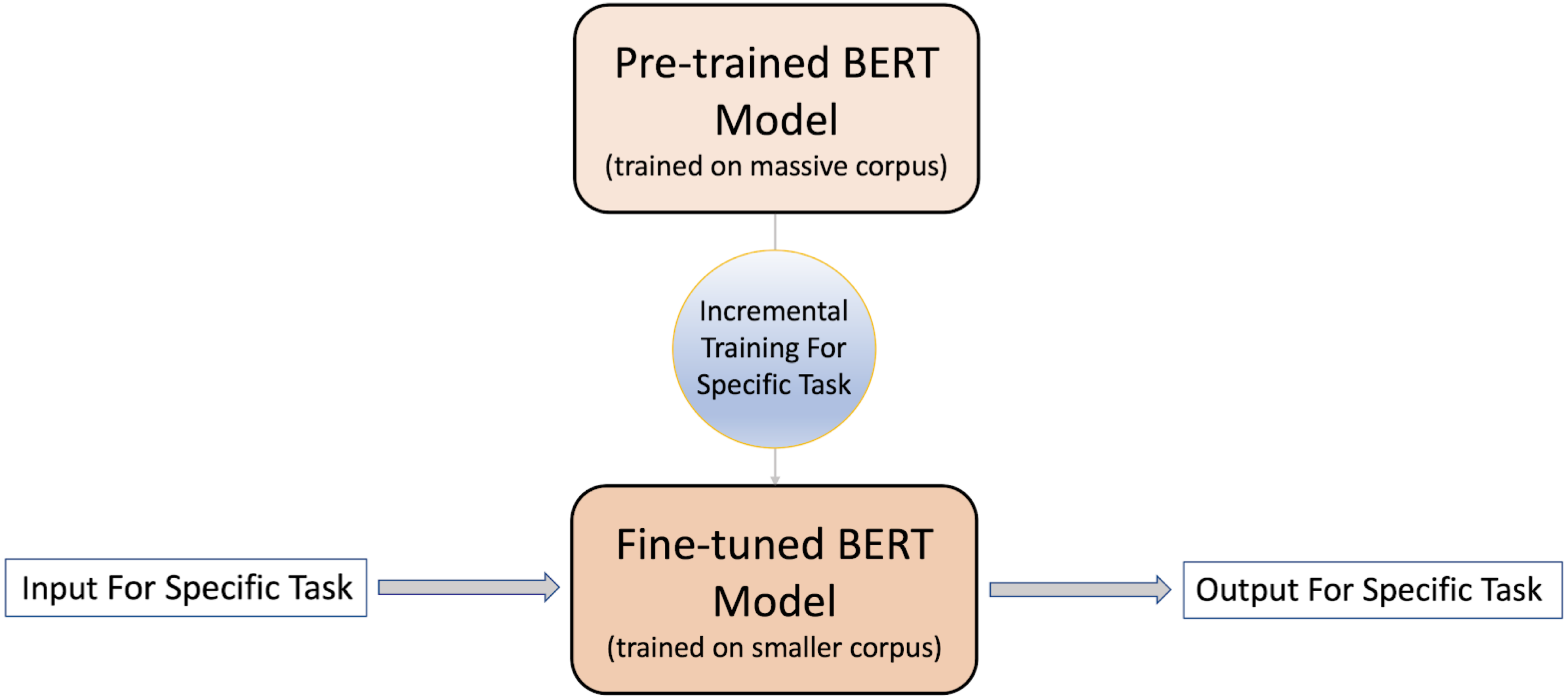


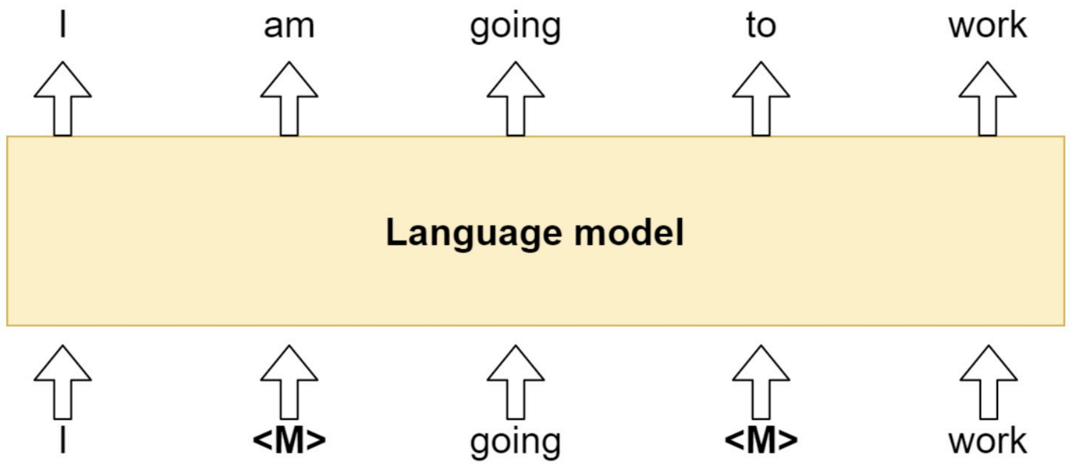
Figure 1: The Transformer - model architecture.

Transfer Learning (BERT)



Transfer Learning cont.

- Pretraining Task – Masked Language Modeling (MLM):



- Downstream Fine-tuning Tasks:
  - Sentiment Analysis
  - Summarization
  - Question & Answering
  - Machine Translation
  - …

Advantage of Transfer Learning

- Utilize unlabeled data and need less labeled data
  - Creating labels is labor-intensive
  - Allow us to feed in more data
- Empirically higher accuracy

Research Context

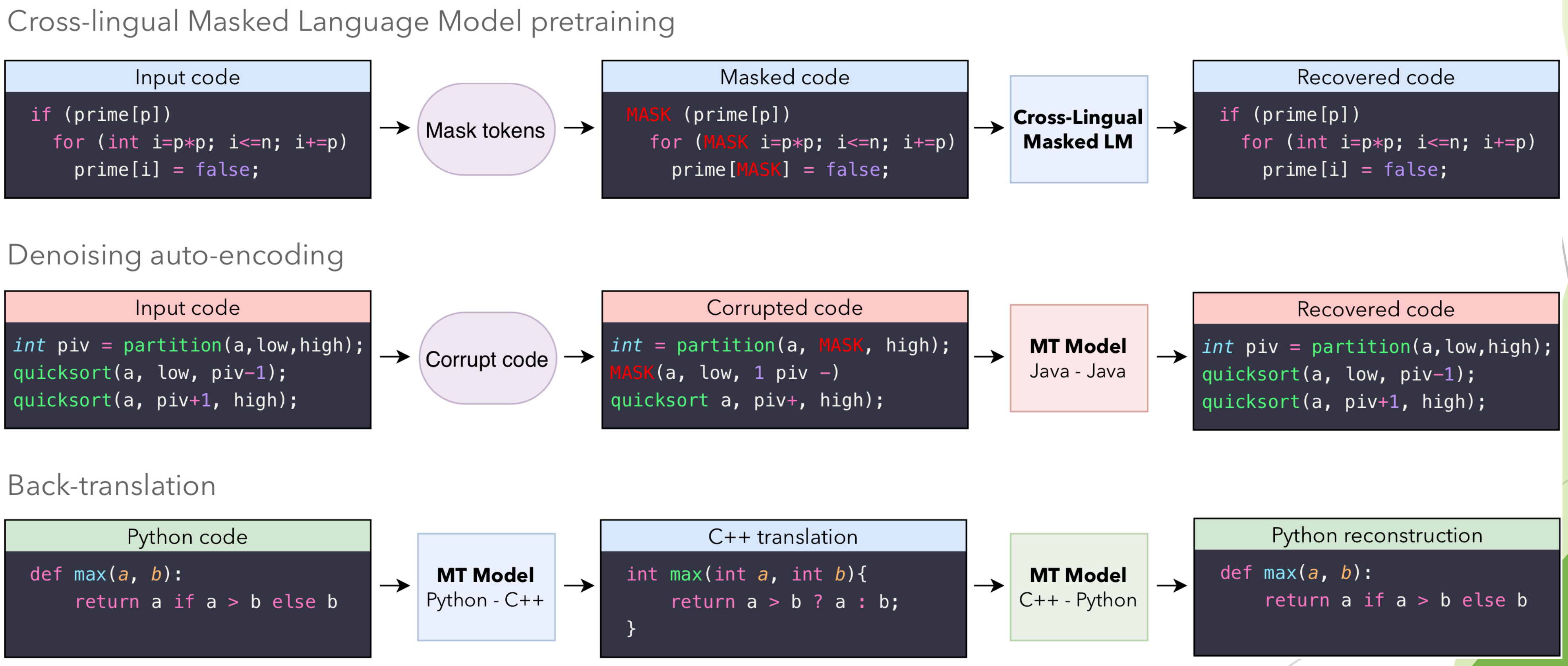
- Transformer has shown success on natural languages
- Transformer has shown success on high-level programming languages
- Previous machine learning models that optimizes compilers never used Transformer before

Our Goal

- Test to see if Transformer language modeling can extract high-level information about the program from low-level programs
- Such information would be able to better inform us where and how to apply compiler optimization

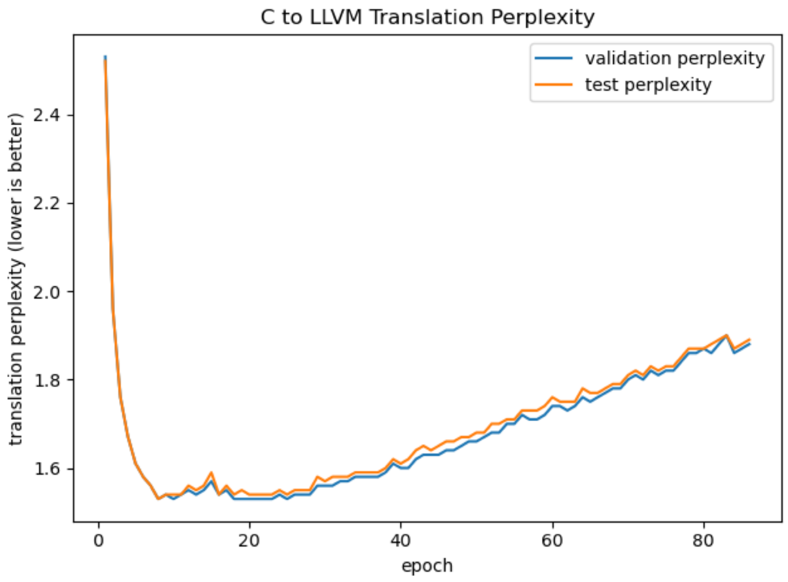
TransCoder (Roziere et al. 2020)

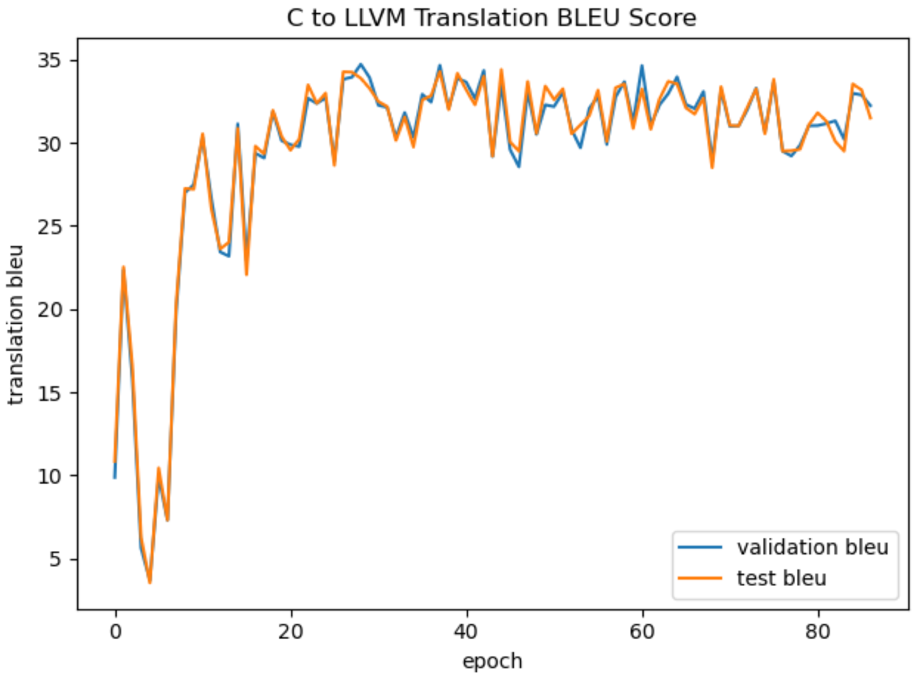
- Unsupervised Translation of Programming Languages

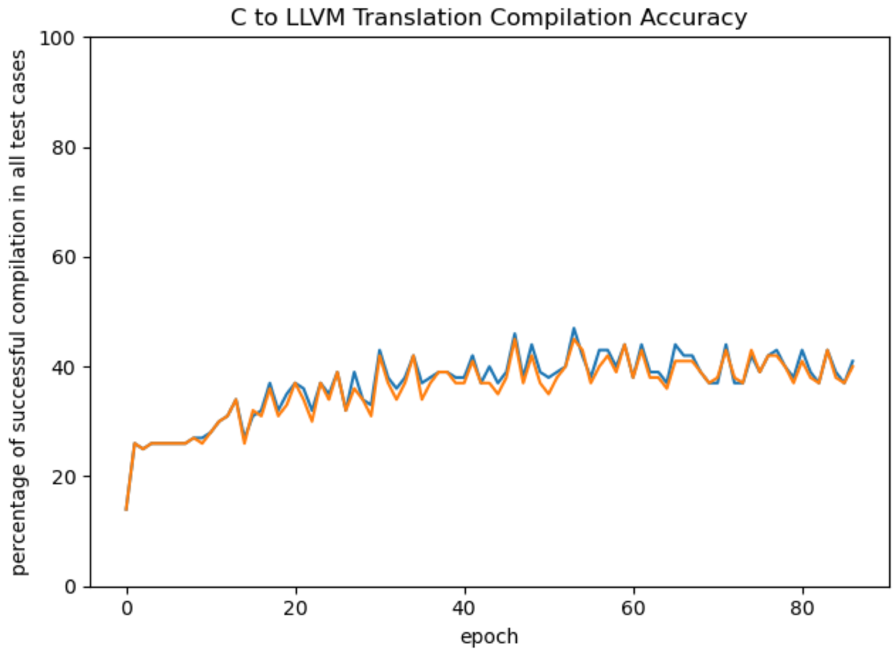


First Case Study: Translating C to LLVM-IR

- Built our own LLVM-IR tokenizer and learned Byte-Pair Encoding (BPE)
- CSmith and CodeNet dataset for C code and generated their LLVM-IR counterparts
- Pretrained with MLM on all data but only fine-tuned on functions
- Overfitting





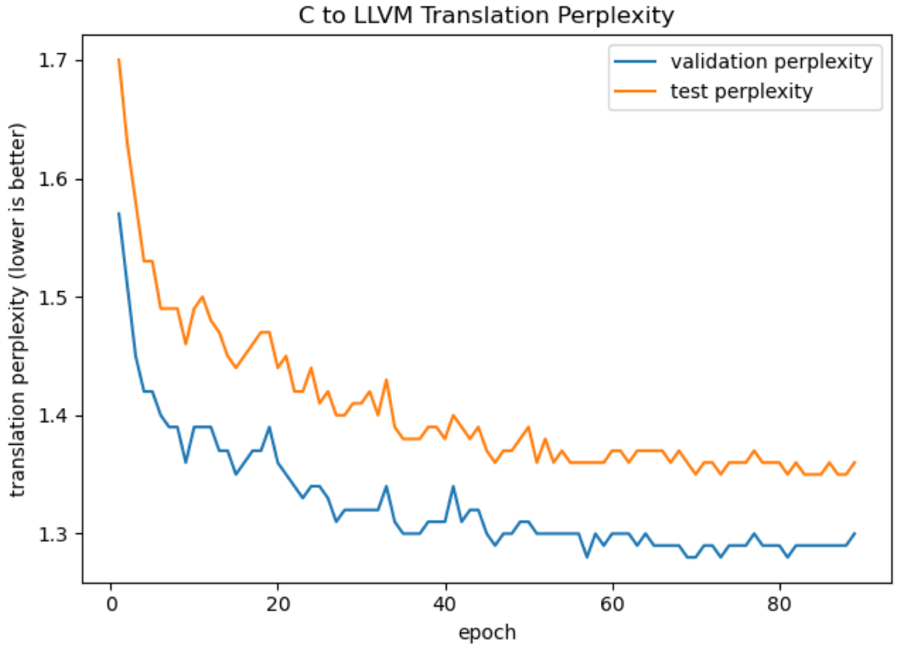


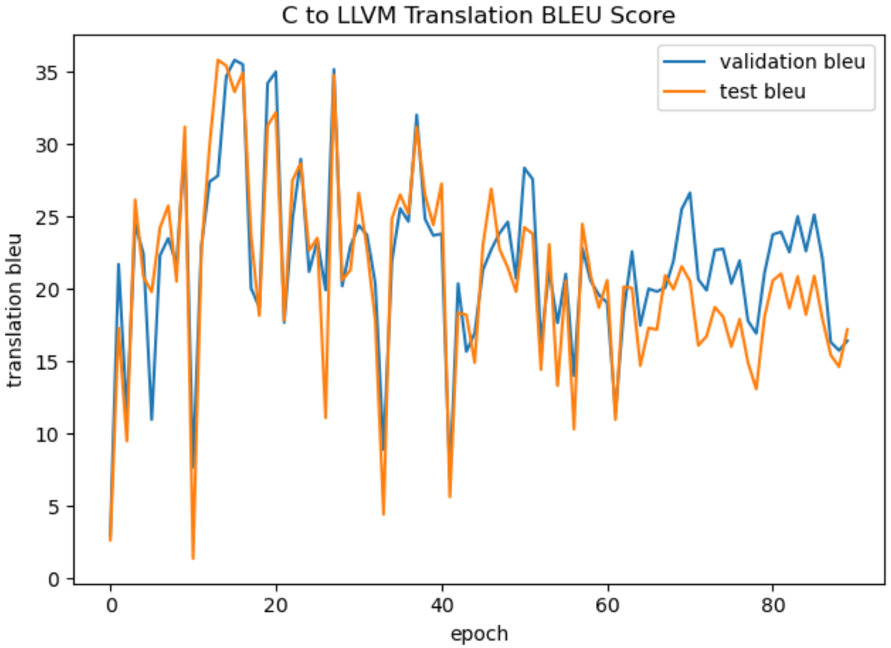
TransCoder	BLEU
C++ → Java	85.4
C++ → Python	70.1

Modifications

- Removing unnecessary syntax while making sure it compiles
- Prefix Notation
  - A * B + C / D = + * A B / C D
  - Based on a paper on Transformer performing arithmetic, prefix notation performs better than infix or postfix notion, yielding a 94.43 BLEU Score comparing to 87.72 for infix and 92.37 for postfix.
  - { 8, [3, 5.0, Carl] } = STRUCT2 8 ARR3 3 5.0 Carl
- Masking variables to be have random names rather than %1, %2, %3…

Performed better but still not satisfactory…





Fundamental Challenges

- Low-level language highly repetitive → MLM performing less well
  - New code-specific pretraining objective
- Low-level language needs longer length of sequences to represent a short sequence in high level language
- Low-level language assumes too much information unknown to the high-level language
- Doesn’t perform well with high-level tasks that interact with high-level languages

Second Case Study: Throughput Estimation of X86_64 Basic Blocks

- Basic block = chunks of assembly code without branches
- Throughput = clock cycles for executing a basic block in steady state
- Accurate throughput estimation is an essential tool that informs choosing the proper optimization passes
- Ithemal (Mendis et al. 2018) uses a hierarchical LSTM for its estimation
- Some struggles but overall shown better results than code translation

DynamoRIO Tokenizer

- Recovers hidden information in the Intel syntax
  - $mul\ ecx\ \ = mul\ eax\ ecx, edx\ eax$
- Automatically gets rid of unnecessary syntax, such as brackets and memory displacements
  - The LLVM-IR tokenizer would recognize brackets as separate tokens
- No need to BPE because vocab is already small

```
sum:
        .cfi_startproc
# %bb.0:
        push        rbp
        .cfi_def_cfa_offset 16
        .cfi_offset rbp, -16
        mov         rbp, rsp
        .cfi_def_cfa_register rbp
        mov         dword ptr [rbp - 4], edi
        mov         dword ptr [rbp - 8], esi
        mov         eax, dword ptr [rbp - 4]
        add         eax, dword ptr [rbp - 8]
        pop         rbp
        .cfi_def_cfa rsp, 8
        ret
.Lfunc_end1:
        .size        sum, .Lfunc_end1-sum
        .cfi_endproc
```

Throughput Estimation Experiment

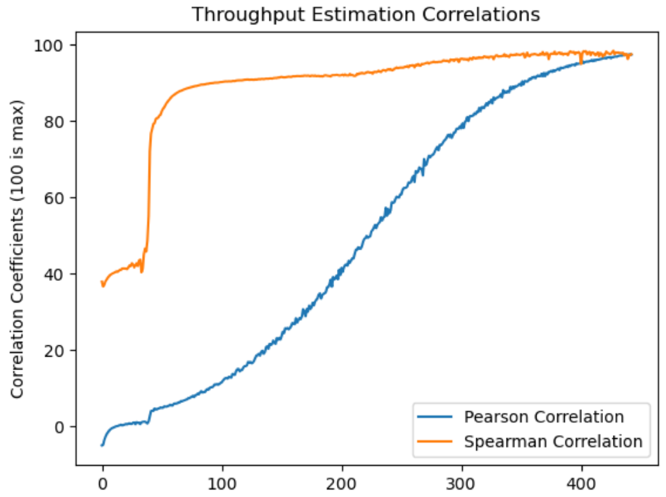
- BHive benchmark dataset with 320,000+ basic blocks mapping to the throughput under Intel's Haswell microarchitecture
  - While the majority of data points fall under value between 20.0 and 1000.0, the maximum can go up to 1,600,450
- Pretrained on MLM and fine-tuned with MSE loss for regression on the same dataset

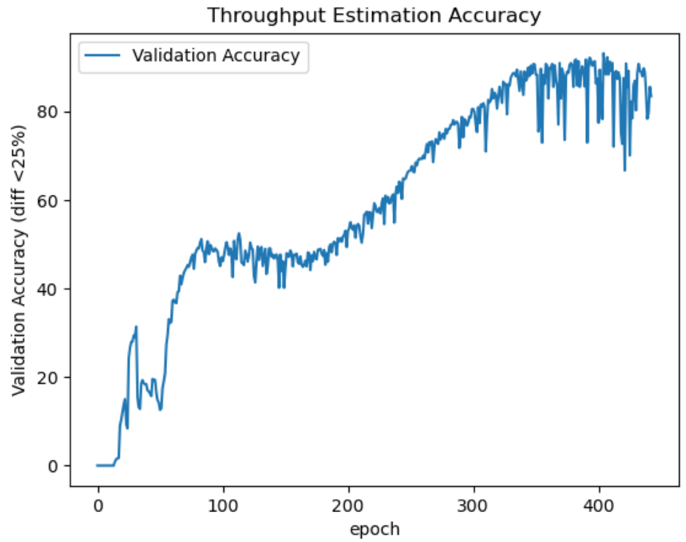
```
mov    rdx, qword ptr
[rbx+0x50]
xor    ecx, ecx
mov    esi, 0x01178629
mov    rdi, rbp
```

→ 110.00

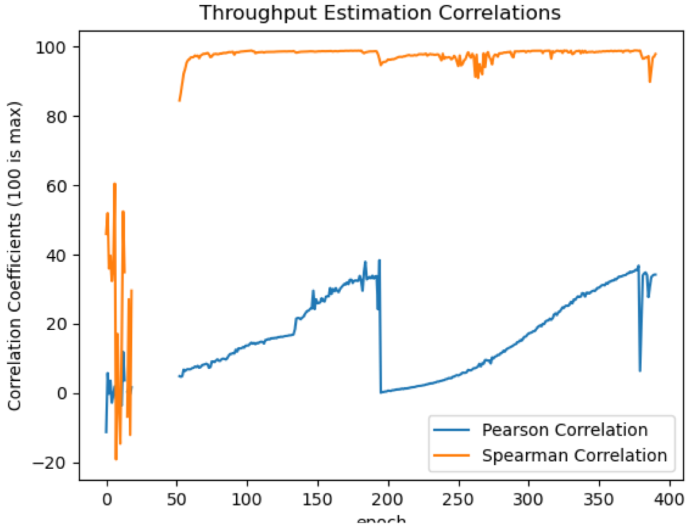
1,000 Case Proof of Concept

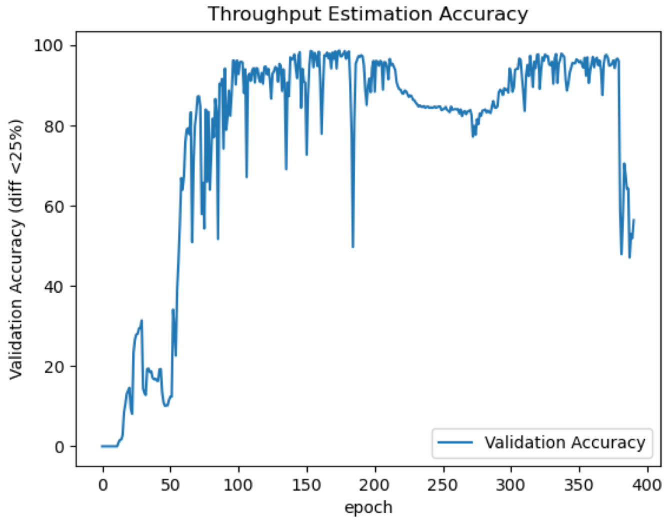
Fine-tuning projection layer





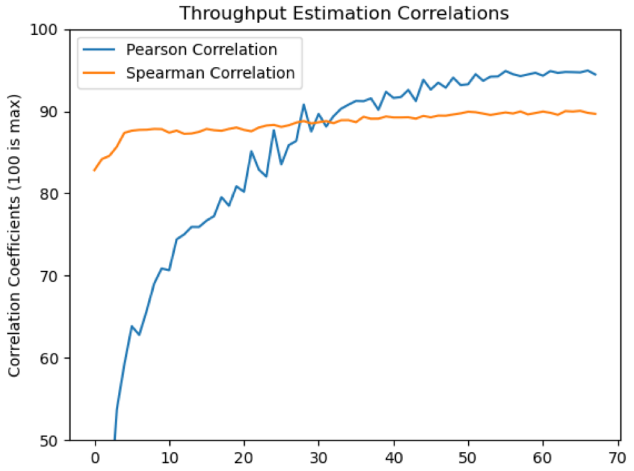
Fine-tuning projection layer and embedding space

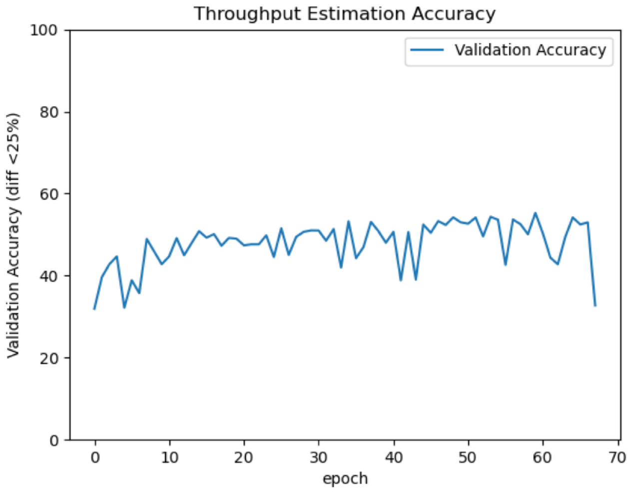




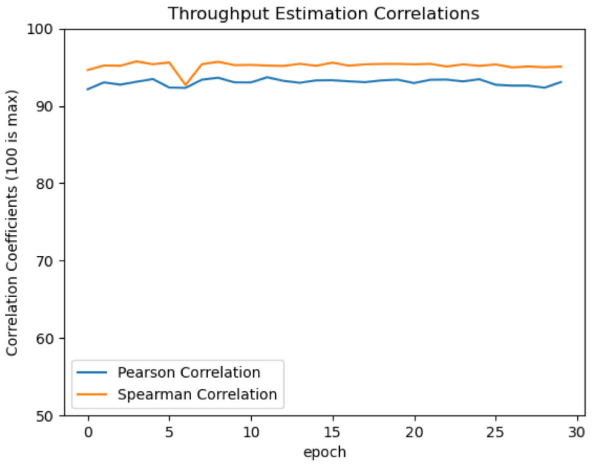
Full dataset

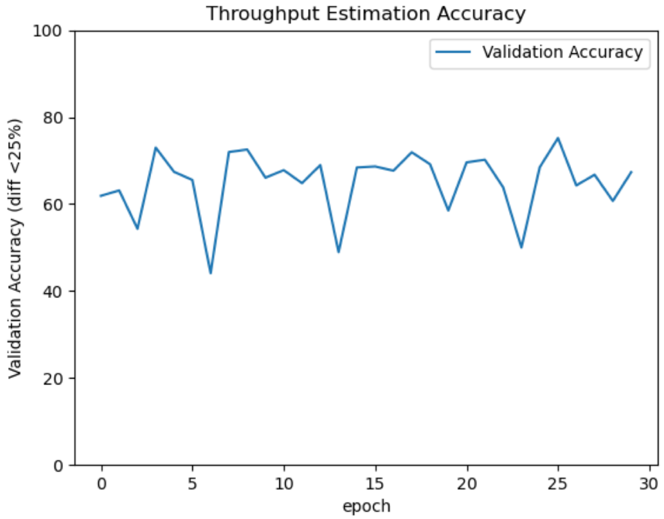
Fine-tuning projection layer



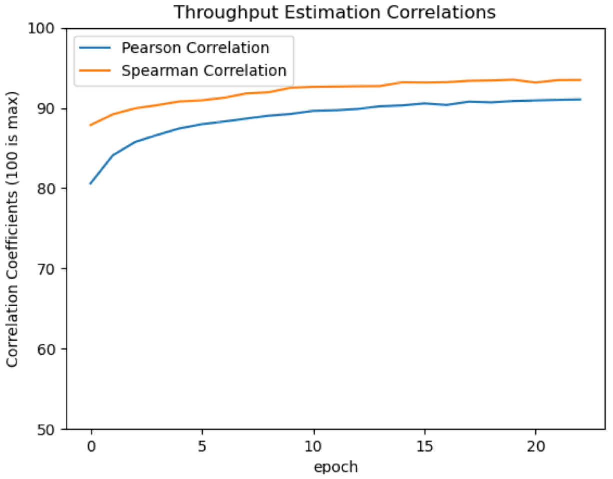


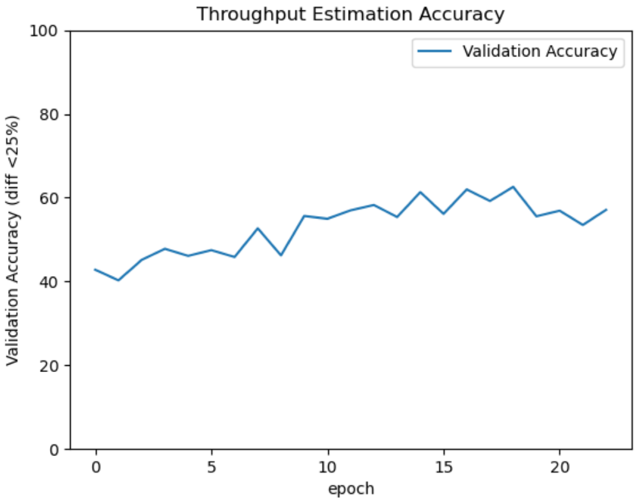
Fine-tuning projection layer & embedding
With lab2id





Fine-tuning projection layer
With lab2id





Observations

- Both Ithemal and Transformer struggle with large values
- Lab2id tries to mitigate the issue
- The model has to truncate sequences to a max length of 512
- While Ithemal can be more exact for the small data points but is really far off for these big outliers, Transformer seems to model the big data points better but be less exact for all data points.

	Pearson Correlation	Spearman Correlation
Ithemal	91.8	96.0
Proj. Layer	94.95	90.04
Proj. Layer with Lab2id	91.15	93.6
Proj. Layer & Embedding with Lab2id	93.69	95.74

Some Prediction Examples

Fine-tuning projection layer
With lab2id

Predicted	Actual
53.0	49.0
345.0	301.0
1779.0	1697.0
3287.5	3087.5
61.0	59.0
2481.25	2295.0
111.0	100.0

Fine-tuning projection layer
& embedding with lab2id

Predicted	Actual
56.0	49.0
277.0	301.0
1479.0	1697.0
3107.0	3087.5
61.0	59.0
2415.0	2295.0
100.0	98.0

My reproduction of Ithemal

Predicted	Actual
33.02	33.00
99.13	98.00
309.76	304.00
31.38	31.00
139.45	1400.00
70.00	399.00
644.00	2295.00

Future Plans to explore

- More data to saturate the model
- Other pretraining objectives (NSP, Denoising)
- More ablation studies on tokenization

Conclusion

- Transformer on low-level languages has shown more success on low-level tasks than on high-level tasks
- MLM and traditional NLP tokenizers might not perform that well
- DynamoRIO, as an assembly language specific tokenizer, is more helpful
- Despite current struggles, Transformer shows strong potential for future usage in the compiler optimization genre

Acknowledgement

- My Mentor, Billy Moses, for his tireless support
- MIT PRIMES, for this incredible opportunity
- My parents
- All of you, for listening

Questions?

Thank you!