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
Series of Transformation

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

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

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

- **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:

```assembly
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){
    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: $\Theta(n)$
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
Transfer Learning (BERT)

Pre-trained BERT Model
(trained on massive corpus)

Incremental Training For Specific Task

Fine-tuned BERT Model
(trained on smaller corpus)

Input For Specific Task → Incremental Training For Specific Task → Fine-tuned BERT Model → Output For Specific Task
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

Cross-lingual Masked Language Model pretraining

<table>
<thead>
<tr>
<th>Input code</th>
<th>Mask tokens</th>
<th>Masked code</th>
<th>Cross-Lingual Masked LM</th>
<th>Recovered code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (prime[p]) for (int i=p; i&lt;=n; i+=p) prime[i] = false;</td>
<td></td>
<td>MASK (prime[p]) for (MASK i=p; i&lt;=n; i+=p) prime[MASK] = false;</td>
<td>Cross-Lingual Masked LM</td>
<td></td>
</tr>
</tbody>
</table>

Denoising auto-encoding

<table>
<thead>
<tr>
<th>Input code</th>
<th>Corrupt code</th>
<th>Corrupted code</th>
<th>MT Model Java - Java</th>
<th>Recovered code</th>
</tr>
</thead>
<tbody>
<tr>
<td>int piv = partition(a,low,high); quicksort(a, low, piv-1);</td>
<td></td>
<td>int = partition(a, MASK, high); MASK(a, low, 1 piv -) quicksort a, piv+, high;</td>
<td>MT Model Java - Java</td>
<td></td>
</tr>
</tbody>
</table>

Back-translation

<table>
<thead>
<tr>
<th>Python code</th>
<th>MT Model Python - C++</th>
<th>C++ translation</th>
<th>MT Model C++ - Python</th>
<th>Python reconstruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>def max(a, b): return a if a &gt; b else b</td>
<td></td>
<td>int max(int a, int b){ return a &gt; b ? a : b; }</td>
<td>MT Model C++ - Python</td>
<td>def max(a, b): return a if a &gt; b else b</td>
</tr>
</tbody>
</table>
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
  - $A \times B + C \div D = + \times A B \div C D$
  - 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.
  - $\{8, [3, 5.0, Carl]\} = \text{STRUCT2} 8 \text{ ARR3} 3 5.0 \text{ Carl}$
- 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
  - \texttt{mul \textit{ecx} = mul \textit{eax} \textit{ecx}, \textit{edx} \textit{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
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

```assembly
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.

<table>
<thead>
<tr>
<th></th>
<th>Pearson Correlation</th>
<th>Spearman Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ithemal</td>
<td>91.8</td>
<td>96.0</td>
</tr>
<tr>
<td>Proj. Layer</td>
<td>94.95</td>
<td>90.04</td>
</tr>
<tr>
<td>Proj. Layer with Lab2id</td>
<td>91.15</td>
<td>93.6</td>
</tr>
<tr>
<td>Proj. Layer &amp; Embedding with Lab2id</td>
<td>93.69</td>
<td>95.74</td>
</tr>
</tbody>
</table>
## Some Prediction Examples

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>53.0</td>
<td>49.0</td>
</tr>
<tr>
<td>345.0</td>
<td>301.0</td>
</tr>
<tr>
<td>1779.0</td>
<td>1697.0</td>
</tr>
<tr>
<td>3287.5</td>
<td>3087.5</td>
</tr>
<tr>
<td>61.0</td>
<td>59.0</td>
</tr>
<tr>
<td>2481.25</td>
<td>2295.0</td>
</tr>
<tr>
<td>111.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

### Fine-tuning projection layer

With lab2id

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>56.0</td>
<td>49.0</td>
</tr>
<tr>
<td>277.0</td>
<td>301.0</td>
</tr>
<tr>
<td>1479.0</td>
<td>1697.0</td>
</tr>
<tr>
<td>3107.0</td>
<td>3087.5</td>
</tr>
<tr>
<td>61.0</td>
<td>59.0</td>
</tr>
<tr>
<td>2415.0</td>
<td>2295.0</td>
</tr>
<tr>
<td>100.0</td>
<td>98.0</td>
</tr>
</tbody>
</table>

### Fine-tuning projection layer & embedding with lab2id

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.02</td>
<td>33.00</td>
</tr>
<tr>
<td>99.13</td>
<td>98.00</td>
</tr>
<tr>
<td>309.76</td>
<td>304.00</td>
</tr>
<tr>
<td>31.38</td>
<td>31.00</td>
</tr>
<tr>
<td>139.45</td>
<td>1400.00</td>
</tr>
<tr>
<td>70.00</td>
<td>399.00</td>
</tr>
<tr>
<td>644.00</td>
<td>2295.00</td>
</tr>
</tbody>
</table>

### My reproduction of Ithemal

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.02</td>
<td>33.00</td>
</tr>
<tr>
<td>99.13</td>
<td>98.00</td>
</tr>
<tr>
<td>309.76</td>
<td>304.00</td>
</tr>
<tr>
<td>31.38</td>
<td>31.00</td>
</tr>
<tr>
<td>139.45</td>
<td>1400.00</td>
</tr>
<tr>
<td>70.00</td>
<td>399.00</td>
</tr>
<tr>
<td>644.00</td>
<td>2295.00</td>
</tr>
</tbody>
</table>
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!