PRIMES Final Research Paper
January18, 2022

The Effectiveness of Transformer Models for Analyzing Low-Level Programs

Zifan (Carl) Guo
Mentor: William S. Moses
St. Mark's School
Southborough, MA 01772
carlguo866@gmail.com

Abstract

Recently, transformer networks have enabled breakthroughs in the field of natural language processing. This is partially due to the fact that transformer models can be first trained on a large corpus of unlabeled data prior to fine-tuning on a downstream task. Unlike natural language, which is somewhat tolerant of minor differences in word choices or ordering, the structured nature of programming languages means that program meaning can be completely redefined or be invalid if even one token is altered. In comparison to high-level languages, low-level languages are less expressive and more repetitive with more details from the computer microarchitecture. Whereas recent literature has examined how to effectively use transformer models on high-level programming semantics, this project explores the effectiveness of applying transformer models on low-level representations of programs that can shed light on better optimizing compilers. In this paper, we show that transformer models can translate C to LLVM-IR with high accuracy, by training on a parallel corpus of functions extract from 1 million compilable, open-sourced C programs (AnghaBench) and its corresponding LLVM-IR after compiling with Clang. We also present another case study that analyzes x86_64 basic blocks for estimating their throughput. We discuss various changes in data selection, program representation, network architecture, and other modifications that influence the effectiveness of transformer models on low-level programs.

1 Introduction and Related Work

Historically, programmers could rely on the continuous performance improvements stemming from Moore's Law as a way to forgo performance engineering. However, as we enter the *post-Moore's Law* era [9], the development of free performance optimizations from general hardware alone is reaching a plateau, and programmers must turn elsewhere for performance, such as the use of a compiler to automatically optimize code. Merouani et al. [23] pointed out that an optimized implementation of a deep learning neural network, such as XLNet [38], is $1.8\times$ faster than the conventional PyTorch implemented counterpart. Such a speed boost in deep learning model training can greatly reduce the number of resources devoted to running the program and, thus, reduce cost, whether it is for scientific research or for commercial uses.

Consider the code snippet in Figure 1 that normalizes a vector. The *loop invariant code motion (LICM)* [24] optimization pass can reduce its computation time from $\Theta(n^2)$ to $\Theta(n)$ by moving the `map` function out of the loop. Selecting the right optimization pass can significantly improve the program's performance.

This need for optimization and for less time-consuming, better-performing compilers is especially true with complex, large programs. The larger the program, the lower the possibility that computer

Preprint. Under review.

```
__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;
    }
}
```

Figure 1: The left shows the original program to normalize a vector. The `norm` function would normally compute in $\Theta(n^2)$ time without any optimization because the `map` function computes in $\Theta(n)$ time. However, a compiler with a *loop invariant code motion (LICM)* [24] would move the map function outside of the loop and result in the code shown on the right that computes in $\Theta(n)$ time.

scientists could manually tune the program for better efficiency. Moreover, the optimization resulting from manual tuning is specific to the target architecture and cannot provably optimize programs for all targets. This untransferrable nature posts a barrier in generalizing performance from one to all target architectures, which is essential when one wants to achieve the similarly optimized performance on another hardware setup. This obstacle, in turn, fuels the development of automatic compiler optimization that utilizes machine learning.

While this field of study grows at a fast pace, most of the existing literature relies on the structured nature of low-level programs within compilers and trains machine learning models based on extracted, structural information and usually does so in a supervised manner. We, however, attempt to tackle the compiler optimization problem by leveraging recent breakthroughs in unsupervised machine learning, namely transformer models. We treat low-level languages, specifically LLVM intermediate representation (IR) and x86_64 assembly, syntactic token by token, as if they are natural languages, such as English or Portuguese, or high-level programming language, such as C or Java, on which the transformer models have found recent success. In one case study, we focus on the translation between C and LLVM-IR. We built pre-trained models using the Masked Language Modelling objective [8] and fine-tuned it on parallel corpora of C to unoptimized LLVM-IR datasets, receiving proofs that the transformer model can successfully translate C functions into LLVM-IR. We find similar results when we try to translate C to LLVM-IR optimized with `-O1` flag. Such success attempts reflect the ability of transformer models to understand the inner workings of LLVM-IR language syntax, despite its repetitiveness, and sheds light on the future possibility of applying transformer architecture to provide better optimization than standard optimization flags. Our work shows that the selection of the right C compilation dataset is the key to transformers' performance, and reducing the repetition within unoptimized LLVM-IR programs and presenting data in prefix notation can help the model to perform better in translation. In another case study, we attempt to renovate Mendis et al. [22] 's hierarchical LSTM model that estimates throughput of x86_64 basic blocks with transformer and yields results that match state of the art.

1.1 Compilers

Compilers translate source code in a high-level programming language like C or C++ to lower-level languages. High-level languages are portable, readable, often with high abstraction and elements of natural language like English. Lower-level languages, being closer to the hardware, lack readability but are necessary for the hardware to present executable results. Moreover, they are highly optimized as they are designed specifically for a particular type of targeted hardware. Compilers are critical to enabling programmers to code at ease without worrying about the lower-level machine code's lack of readability and hardware specificity. It connects software to its target software.

Compilers, such as LLVM [20], first produce an intermediate representation (IR) of the source code before transforming it to lower-level executables to achieve the best of both worlds. It can withhold complete source code information and is independent of both the source languages and targeted hardware. The intermediate representation allows for more portability in generating better optimization,

which is usually implemented with a sequence of different optimizing code transformations that result in semantically equivalent IRs that run faster.

1.2 Compiler Optimization

The problem of compiler optimization has existed in the field of computer science for decades. Although a myriad of problems exists in the process of optimization, Ashouri et al. [2] identified two main issues for better optimization: optimization selection and phase-ordering. The former focuses on what content of optimization to adopt. The latter surrounds the order of applying the chosen optimizations. The latter exists because sometimes a particular optimization pass `A` would transform the code in a way that might "hinders the effect of some optimizations that otherwise could have been performed by the following pass `B`"[2]. On the other hand, having a particular transformation before another might lead to better performance of the latter.

Existing compilers contain numerous preconstructed code transformation passes; for example, Clang, the LLVM frontend for C and C++, has more than 150 transformations, such as the *loop invariant code motion (LICM)* mentioned earlier, loop tiling, and inlining. The sheer number of code transformation passes also makes the task of constructing heuristics difficult. The problem of optimization lies in choosing a sequence of right code transformations from the repertoire in the right order. Many of such code optimizations are dependent on the programming language or architecture, and aggressively applying these code transformations might result in worse performance than unoptimized ones [2]. Therefore, it is crucial to select the right sequence and, hence, the need to apply machine learning. Merouani et al. [23] concluded that the state-of-the-art compilers had shown success in solving the first problem, but work needs to be done to tackle the second problem successfully.

Many of the existing compilers provide standardly named optimization flags, such as `-O1`, `-O2`, `-O3`, or `-Os`, that are arbitrarily handpicked by compiler designers to apply a particular set of transformations in a particular order. They offer a set of simple choices for the developers that trade-off compilation time with run time, while a higher level is generic more optimized for most programs. It is up to users themselves to determine which flag would result in a more optimized compilation for their particular programs. Even though they might provide sufficient functionality for generic users, they are clearly not the best in terms of targeted optimization for their programs. Numerous attempts have been made to generate more specified, better-performing code transformations outside of the scope of preconstructed ones, but this project would only work with the existing ones and aim to learn from them to achieve a result that performs better than the standard ones.

1.3 Compiler Optimization with Machine Learning

There have been several attempts of compilation optimization using machine learning to tackle each of the two essential problems mentioned above. Ashouri et al. [2] split the related work on compiler optimization with machine learning based on how the model characterizes the programs. Some previous attempts select specific static features of the source code or in the compilation process as proxies for the whole programs when training. Some select source-code features, such as the name of the current function, the values of compiler parameters, or the pass ordering in the current run of the compiler, using tools like[11]'s Milepost GCC. The benefit of static feature extraction is that collecting such features doesn't require the code to be executed, which makes the process less resource-intensive and accessible.

Other existing attempts use a more dynamic approach to characterize code through performance counters that provide information on how well the code runs. Cavazos et al. [5] built a machine learning model to predict the set of code optimization sequences based on performance counters, despite that such characterization is usually architecture-dependent. Traditionally, those optimization models using performance counters performed a lot better than those using source-code features, but one needs to run the program multiple times to collect such data.

To achieve a middle ground, some models adopt graph-based features. Park et al. [26] built a novel model for speedup prediction based on the graph-based characterization of the intermediate through control flow graphs (CFG). Doing so maintains a high level of expressiveness as performance counter features but remains "static" like the source-code features. Tools like LLVM's Opt exist to easily extract control flow graphs from a function or a program [2]. Such characterization can still be limited, as they only select parts of a program to feed into the machine learning model for training.

One of the prominent works on code optimization prediction is COBAYN, a Bayesian Network model built by Ashouri et al. [3], looking at static, dynamic, or hybrid features. However, the model is supervised and usually only works to select optimization heuristics, which still lacks portability. Unsupervised works mainly surround genetic algorithms such as Neuro Evolution of Augmenting Topologies (NEAT). Kulkarni et al. used such an algorithm to address both the optimization selection [34] and the phrase ordering problem [17]. Also, Huang et al. [15] developed the AutoPhase model to optimize the phase ordering for HLS compilers with deep reinforcement learning.

With a relatively simpler objective of speedup prediction, Merouani et al. [23] applied deep learning to build a cost model implemented in the Tirasimu compiler, trying to tackle phase-ordering. Merouani et al. [23] addresses two problems in previous cost models: that it only applies to basic assembly blocks instead of full programs and that it is heavily engineered. However, the model only assesses a few code transformations revolving around loops, such as loop fusion or tiling. Mendis et al. [22] developed an effective hierarchical LSTM model, Ithemal, for a similarly narrow question, estimation of throughput given x86_64 assembly basic blocks.

Previous deep learning works that predict compiler sequences seem to be sparse, and most of the attempts aimed to solve a part of the problems and have their limitations. Our projects aim to look at the optimized intermediate representation on a language level, i.e., all of the available static features, which serves as a more holistic approach than the current literature. Our work shows the potential and serves as a first step towards utilizing transformer models to tackle both problems, optimization selection and phase-ordering, at once.

1.4 Unsupervised Machine Translation

Meanwhile, the field of natural language processing (NLP) has gained heat in the past few years. Recent developments in NLP include the transformer model by Vaswani et al. [35] that performed exceptionally well in language translation tasks, such as translating sentences from Portuguese to English. Specifically, the transformer model revolutionizes the traditional Recurrent Neural Network (RNN) model that clusters layers of feedforward neural network with previous outputs as inputs for the next layer. RNN is known to be slow to train and suffers from the vanishing gradients problem in long data sequences. Attempts such as LSTM [14] try to solve vanishing gradients but still remain slow because the process is still strictly sequential. transformer model yields better results and updates on these networks by training sequences in parallel through the concept of attention that calculates the relevance of each word in a sentence to each other to itself, which, most importantly, can be done while disregarding their distances in the input or output sequence.

Multiple attempts advanced state of the art for NLP tasks since the publication of the original transformer paper, including the BERT model [8] for monolingual pretraining and later the Cross-lingual Language Pretraining Model (XLM) [18]. BERT establishes Masked Language Modeling (MLM) as an effective pre-train objective. XLM upgrades the BERT model to better perform translation between multiple languages with training objectives like Back-Translation, first established by Lample et al. [19]. It also adds the Byte-Pair Encoding (BPE) [33] to increase the shared vocabulary between languages that BERT lacks. The input data can only bring a fixed vocabulary, but translation should be an open-vocabulary problem in real life. BPE splits words into sub-words so the program can better deal with these potential rare and unknown words that do not previously exist in the vocabulary, bettering the performance of BERT on cross-lingual translation [19].

1.5 Unsupervised Machine Translation on Code

While transformer models show consistent, positive results regarding natural languages, researchers also applied transformer models to translate between programming languages in the recent two years. Kanade et al. [16] developed a BERT model to obtain contextual embedding of Python source code, training with five classification tasks. Feng et al. [10] presented CodeBERT, a pre-trained model aiming to capture the semantic similarity between natural and programming languages. They showed that pretraining could improve the performance of downstream tasks like code searches and documentation generation. The most prominent and relevant work is Roziere et al. [30] 's TransCoder, an unsupervised model that translates C++, Java, and Python 3 to each other based on open-sourced GitHub monolingual source code data accessed through Google BigQuery[1].

[1]https://console.cloud.google.com/marketplace/details/github/github-repos

After TransCoder's initial success, Roziere et al. [31] developed DOBF [31] and self training TransCoder (TransCoder-St) [32]. The former is a novel deobfuscation pretraining objective (DOBF) specifically designed for programming languages to replace the popular MLM objective. It asks to recover obfuscated variable and function names, such as VAR1 or FUNC1, based on their actual meaning. Since codes, unlike natural languages, are more structured, "if a given variable appears multiple times in a function," the transformer model applied on programming languages might be able to cheat when training for MLM because it can "simply copy its name from one of the other occurrences" [31]. As a result, MLM does not work very well with repetitive codes, and DOBF can provide a more informative pre-trained model because it is a more difficult task than MLM and bypasses this common error that the machine learning model can perform during MLM. TransCoder-ST [32] identifies the importance of parallel data corpus to unsupervised machine translation. Due to the inherent difficulty of collecting parallel data between C++, Java, and Python, TransCoder [30] settles for monolingual data and fine-tuning on the back-translation objective, which involves training on noisy inputs. TransCoder-ST builds a parallel data corpus by taking an already trained TransCoder to generate predicted translation and leveraging an automated unit-test tool to filter out invalid prediction, then continues to fine-tune the model with this parallel data corpus. Both the DOBF pretraining objective and the generated parallel data corpus for self-training improve upon the state-of-the-art first established by TransCoder.

Since the development of well-designed benchmarks on machine learning for programming languages, such as CodeXGLUE [21], many more attempts exist to apply Transformers to perform various downstream tasks that analyze code and sometimes involve natural language to programming language interactions, including but not limited to code completion, code translation, code generation, and code summarization. Phan et al. [27] developed CoTexT to apply the popular T5 model, or Text-to-Text Transfer Transformer, [29] on code, allowing the model to perform multiple downstream tasks with one model. Ahmed and Devanbu [1] highlighted that different implementations of the same code in multiple programming languages can particularly preserve identifiers naming patterns well, which can serve as an anchor point for training and amplifying performance. GitHub Copilot[2], powered by OpenAI Codex, can provide accurate suggestions to complete lines or whole functions based on the documentation comment and other contexts.

2 Model

To translate C to unoptimized LLVM-IR, we follow the model structure and code implementation established in TransCoder [30]: a sequence-to-sequence (seq2seq) transformer model with attention that consists of an encoder and decoder [3]. The TransCoder model follows the three principles first set out by XLM [18] for cross-lingual natural language translation: initialization, language modeling, and back-translation. We follow the first two steps accordingly but adapt the Machine Translation objective rather than back translation, pretraining with the MLM objective on all the C and LLVM data and training with denoising auto-encoding and back-translation objectives only on the standalone, static function.

2.1 Preprocessing

Then, to process into the ML pipeline, we use separate tokenizers for C and LLVM-IR similar to Roziere et al. [30] because different languages use keywords for drastically different meanings. For example, `";"` indicates the end of one line in C but indicates the start of a comment in LLVM-IR. Facebook researchers originally implemented the C tokenizer using a Python binding of Clang, but later switched to Tree-sitter[4] in their newly updated CodeGen GitHub repository[5]. The two tokenizers function slightly differently, but both accomplish the desired task properly; for example, for the hashtag function definition `#define`, Clang would tokenize it into two tokens `#` and `define` respectively, while Tree-sitter keeps it as one token. We chose to use the Clang C tokenizer because of its internal logic's similarity to the LLVM-IR tokenizer that we implemented, which utilize similar

[2] https://copilot.github.com/

[3] https://github.com/facebookresearch/TransCoder

[4] https://tree-sitter.github.io/tree-sitter/

[5] https://github.com/facebookresearch/CodeGen

libraries. We extended the LLVM library using `PyBind11` [6] to access the `LLLexer` as our LLVM tokenizer. It provides the token types, and we can parse out the string representation of the tokens correspondingly. We then learn BPE codes on these tokens concatenated together, using fastBPE[7], and split them into subword units.

2.2 Cross Programming Language Pretraining

Lample et al. [19] concluded the importance of pretraining in unsupervised machine translation by mapping similar sequences with similar meanings together regardless of the languages. Roziere et al. [30] identified the cross-lingual nature of the pretraining model comes from the number of common tokens (anchor points), such as shared keywords like `define`, variable names, and digits. We believe that the task of translating from C to LLVM inherently presents worse cross-lingual representation than a translation between two high-level languages, such as C++ and Java. This difference is because of the higher syntactical and structural difference between C and LLVM, similar to the logic that an English-French model would have more "cross-linguality" than an English-Chinese model because of the similar alphabet [30]. There should be enough anchor points to consider the C-LLVM model as cross-lingual, but unexplored specifics still exist to form a conclusion with higher certainty.

For the specific pretraining objective, we use the masked language model (MLM) objective [8] following Roziere et al. [30]. Namely, it takes in a text sequence at each iteration, masks out some tokens, and asks the model to predict the missing tokens based on their context. While the deobfuscation pretraining objective, DOBF [31], has been proven to be a more effective pretraining model, we chose not to use DOBF. The original study only implemented DOBF on Java and Python, not C++, because an open-sourced obfuscator for C++ does not exist to generate the obfuscated data the DOBF pretraining requires. We did not have enough time or resources to implement one ourselves at this point and chose not to use it for C programs. Moreover, since LLVM-IR does not generate variables with meaning names but rather `%0`, `%1`, `%2`s, pretraining DOBF on LLVM-IR seems impossible.

2.3 Denoising auto-encoding

While the encoder exactly matches the architecture of the pre-trained XLM model, the decoder needs extra parameters on the source attentions, which are randomly initialized following Lample and Conneau [18]. As the decoder has never trained to decode a sequence before, the model trains the encoder and decoder with the Denoising Auto-Encoding (DAE) objective, which asks the model to predict the sequence of tokens based on a corrupted version of it after adding noise established in Lample et al. [19]. The noise is generated by randomly masking, removing, and shuffling tokens in the input sequences. This step trains the encoder to be robust against noise so that it can better perform the latter back-translation objective [30].

2.4 Machine Translation

With the pretraining MLM and denoising auto-encoding objectives, the model would be able to generate translation based on the input, but it might be of low performance and be dependent on the inherent and unchangeable "cross-linguality" between languages based on the number of common anchor points [30]. TransCoder [30] and XLM [18] are trained on the back-translation objective to mitigate the problem, which translates the sequence in the source language to the target language and translates it back to the source language, on which the loss function is performed. However, as TransCoder-ST [32] identifies, back-translation is a mediocre solution to the problem that they lack parallel data and have to rely on monolingual data because back-translation is less direct than machine translation and creates more noise along the way. If parallel data is available, one should choose machine translation. In the case of translating from C to LLVM-IR, we can easily access such parallel corpus as long as the C program is able to compile.

Machine Translation and Denoising Auto-Encoding train in parallel until they converge.

[6] https://github.com/pybind/pybind11

[7] https://github.com/glample/fastBPE

2.5 Preprocessing Modifications

We first clean up the C data before compiling to generate LLVM-IR with `clang -E`, which writes out all the hashtag functions and library dependencies in C. We also made several attempts to clean out unnecessary parts of the LLVM data that can facilitate better training while ensuring that it would not tamper with the compilation results. This removed information includes target data layout, target hardware architecture, comments, alignments, global attribute groups, and metadata. In some statements, such as `load`, or `getelementptr inbounds`, the type of the data always appears twice, once as itself and another as the pointer to it. In this case, we removed one of the two appearances and made sure our detokenizer could restore it back.

We remove all comments as they are filtered out in the process of compilation and would not provide meaningful information for translation.

At the same time, because we want to eventually compile the translated hypotheses but only train on the level of functions instead of the whole file, some information is inevitably lost in the process and cannot be recovered. While some we can restore back an unexpressive global variable definition or function declaration to reach the bare minimum for the program to compile, the definition of any `struct` is permanently lost and would hinder the program's compilation. We can simply replace the references of a non-recursive `struct` with their definitions without losing meaning. For a `struct` like `%struct.S5 = type { i16, i32, i24 }`, we can replace all occurrences of `%struct.S5` with `{ i16, i32, i24 }`. While it adds complexity to the model than translating directly into `%struct.S5` because it needs to make the extra inference, it seems to be a worthwhile sacrifice to make sure the machine learning predictions compile.

Furthermore, for each C representation of a string, LLVM-IR would automatically generate a global string constant with names such as `@.str.1` or `@.str.2`. The string information would be lost when we only extract functions to train. We have tried to mitigate this problem by replacing unexpressive, automatically generated string variable names with the exact content of the string.

For example, the string definition of the following:

```
@.str.1 = private unnamed_addr constant [8 x i8] c"\0Ahello\0A\00"
```

will be changed into a definition like this:

```
@".str.1:\0Ahello\0A\00" = private unnamed_addr constant [8 x i8] c"\0Ahello\0A\00"
```

However, as long as we know the length of the string, we could also fill in random character tokens to make the programs compile, which seems to be a more effective solution because it gives the model an easier task to learn. For other global constants, the call expressions themselves have enough information to reconstruct at least a declaration, which is a bare minimum for the program to compile.

Moreover, the complex `type` variables in LLVM are difficult for the model to learn. For instance, an array in LLVM-IR is defined like `@ptr = [3 x i32] [i32 1, i32 2, i32 3]`, which is a hard syntax for the machine to learn because it has to consider the scope of the array and where the `[]` ends. In a more extreme example, a `struct` with the type `{ [4 x i8], i32, i8, { i8, i32 }, i64 }` is even harder to comprehend. Griffith and Kalita [12] showed that transformer models would do better in solving arithmetic problems when the arithmetic expressions, as the data, are in prefix notation instead of the conventional infix notation. We made similar attempts that remove structures of `[]` or `{}` and write out the `types` in prefix notation, converting the above struct into `STRUCT 5 ARR 3 4 x i8 i32 i8 STRUCT 2 i8 i32 i64`. By recording the length of the struct, the detokenizer can faithfully restore them back to evaluate the model's performance. Representing data in prefix notation is proven to be easier for the transformer model to understand.

3 Experiment

3.1 Training Details

Following Roziere et al. [30] 's TransCoder, we train our model with a transformer of 6 layers, 8 attention heads, with a single encoder and a single decoder for all programming languages. At

training time, we use batches of around 3500 tokens. We use the GELU [13] as an activation function. We add in a 10% dropout rate and a 10% attention dropout rate. We optimize TransCoder with the Adam optimizer and a learning rate of 10^{-4}. We train them on an NVIDIA GeForce RTX 3090 GPU or an A100 TENSOR CORE GPU. It is worth noting that we have fewer computing resources than the researchers producing the work to which this paper is referencing. Such limitations can also make sure that our work on compiler optimization can be realistic and applicable to the vast majority of developers without robust GPUs.

3.2 Training Data

We have considered multiple data sources for our training data, including CSmith [37], Project CodeNet [28], GitHub Google BigQuery [8], and AnghaBench [7].

CSmith by Yang et al. [37] is a randomized test-case generation tool for C programs, built initially to discover unknown compiler bugs. Regarding the state-of-art when it was published in 2011, it could generate random programs that are comparatively more expressive, containing complex code using many C language features. We first attempted our model on CSmith but received poor results due to its randomness, repetitiveness, and complexity, lacking proximity to humanly written code. It only utilizes relatively simple data structures and operations, which might not represent all the C programs. It only utilizes relatively simple data structures and operations, which might not represent all the C programs. The functions are usually too long, and machine learning models work better with shorter sequences.

Project CodeNet provides a set of benchmarks scrawled from two online judge websites, AIZU Online Judge[9] and AtCoder[10]. These websites contain a finite set of questions for which coding enthusiasts could submit solutions, and these solutions span different languages. Project CodeNet's strength is an established set of parallel data spanning different languages, but unfortunately, we only need the C files. As it turned out, the solutions people submit, especially for simpler questions, can be highly similar and does not generalize well to the LLVM language as a whole. On the C level, different syntax exists to implement the exact same function, such as the difference between writing a `for` loop in one line and in multiple lines, or the difference between writing a `while` loop and a `for` loop. However, such a visible difference on the C level disappears on the LLVM-IR level, as long as the C codes attempt to achieve the same functionalities. Our model struggled when training on CodeNet data.

Google BigQuery provides a public crawl to all available GitHub open-sourced repositories; such a scrawl can generate 3 million C files alone. However, because we have next to no knowledge on the libraries dependencies the C files need, only a limited amount of those files can be compiled with natural `Clang` and used for our projects. We eventually decided not to use this dataset for training due to the difficulty of training.

AnghaBench is a benchmark of more than 1 million C functions, with the required minimal C code to compile them. Built by crawling C files on GitHub, the authors extracted individual functions and applied type-inference to reconstruct the missing definitions required to compile them, such as declarations of auxiliary functions, type definitions, etc. AnghaBench has, by far, the most amount of usable data, which can help saturate the model. Having only one extracted function in each file facilitates the model's training, and our model found success on this benchmark dataset.

While we pre-train on all the source code available, we train with DAE and back-translation objection on only the static functions in C and their corresponding LLVM-IR.

3.3 Evaluation

We evaluate our results on three metrics, the training accuracy generated by the loss function, perfect reference matches, the industry convention BLEU [25] score. Reference match refers to the percentage of translation that perfectly matches the ground truth, while the BLEU score is a widely accepted evaluation metric for natural language translation, with 0 as completely different and 1 as exactly the same.

[8] https://console.cloud.google.com/marketplace/details/github/github- repos

[9] https://onlinejudge.u-aizu.ac.jp/home

[10] https://atcoder.jp/

AnghaBench Dataset	Prefix	Prefix & Global	-O1	Original
Testing Accuracy	98.69	99.29	97.87	99.03
Reference Match	22.62	37.82	**38.73**	13.33
BLEU	78.19	**81.91**	77.03	69.21

Table 1: **Results of unsupervised machine translation on the AnghaBench test set.** The first column, labeled "Prefix", illustrates the training result after converting data representation to prefix notations. The second column shows the result after performing prefix notation and converting global variables and structs into their respective definitions. The third column shows the result of training on LLVM-IR optimized with -O1 flag. The fourth shows the training result with no preprocessing modifications to the original LLVM-IR.

	Csmith	CodeNet
Testing Accuracy	90.73	93.66
BLEU	43.39	51.01

Table 2: **Results of unsupervised machine translation on the Csmith and CodeNet test set.** The training on both datasets adopts the best possible preprocessing modifications, but due to the nature of the datasets, the training on AnghaBench significantly outperforms that on Csmith or CodeNet.

TransCoder has to rely on back-translation, evaluating a BLEU score between the original C code and predicted C code after translating twice. However, back-translation might make BLEU score uninformative, because the model can translate into some LLVM-IR gibberish but translate back to proper C. Because generating parallel matching data for C and LLVM-IR isn't as hard, the direct machine translation approach used by our project makes the evaluation of BLEU score more informative. Directly testing whether the programs can compile is another good metric, but the numerous preprocessing modifications we employed made such a detokenization process difficult, an area that is left for our future work.

3.4 Results

The current results are reported in the following tables. We report the results on our AnghaBench test set, with various preprocessing modifications, in Table **??**, and give an example of such unsupervised translation from C to LLVM-IR tested on AnghaBench in Figure 2. We report the results of training on Csmith and CodeNet data in Table 2. We observe that the transformer model performs significantly better on the AnghaBench dataset than on Csmith and CodeNet, due to the dataset's expansiveness and humanly readable syntax. Applying prefix notation transformation, removing redundant language syntax, and writing out global variable and struct definitions within the function also help the model to perform better.

```
mysig_t mysignal ( int sig , mysig_t act ) {
    return ( signal ( sig , act ) ) ;
}
```

```
define dso_local i32 @mysignal ( i32 %0 , i32 %1 ) #0 {
 %3 = alloca i32
 %4 = alloca i32
 store i32 %0 , i32 * %3
 store i32 %1 , i32 * %4
 %5 = load i32 , i32 * %3
 %6 = load i32 , i32 * %4
 %7 = call i32 @signal ( i32 %5 , i32 %6 )
 ret i32 %7
 }
```

Figure 2: **Example of LLVM-IR prediction with the transformer model.** The right is a verbatim match to the excepted compiler output.

	Spearman Corr.	Pearson Corr.	Validation Accuracy (<25% margin of error)
Proj. layer Only	90.04	**94.95**	55.27
Proj. layer & Embedding	89.35	63.73	51.04
Proj. layer$_{label2id}$	95.29	91.95	76.06
Proj. layer & Embedding$_{label2id}$	95.74	93.69	75.19
Replicated Ithemal	**96.0**	91.8	**88.39**

Table 3: **Results of applying transformer model to estimate the throughput of x86_64 basic blocks** The transformer model yields results worse or matching the original Ithemal model. Among the different ablations of transformer models, fine-tuning with projection layer and *label2id* dictionary performs the best, with a validation accuracy of 76.06%, but the difference is almost indistinguishable with the model that fine-tunes on both the projection layer and embedding and mapping throughputs with *label2id*, with a validation accuracy of 75.10%.

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

110.0

Figure 3: The above figure is an example of the data points in the BHive dataset. The left illustrates one x86_64 basic block, and the right shows its corresponding throughput as a numerical value.

4 Throughput Estimation of x86_64 Assembly Basic Block

In another separate case study, we have also attempted to renovate Mendis et al. [22] 's Ithemal, which utilizes a hierarchical LSTM model, with transformer models. Accurate throughput estimation is an essential tool to inform the computer how to choose the proper optimization passes.

4.1 Setup

We have trained our transformer model on the BHive [6] benchmark dataset, with 320,000+ x86_64 basic blocks mapped to throughput when running on the Intel Haswell microarchitecture. An example of the data can be found in Figure 3. We followed the preprocessing structure outlined in Ithemal, adopting a DynamoRIO [4] tokenizer. DynamoRIO recovers hidden information in the Intel syntax; for example, the tokenizer will recover `mul ecx` into `mul eax ecx, edx eax`. Unlike the LLVM-IR tokenizer that recognizes brackets as separate tokens, DynamoRIO can remove unnecessary syntaxes, such as brackets and memory displacements. Furthermore, unlike because assemblies do not contain any English elements, the vocab for assemblies is small (less than 2000 tokens), so there is no need to perform BPE on the assembly basic blocks.

4.2 Results

We pre-trained the model with Masked Language Modelling and fine-tuned it with MSE loss for regression on the same dataset. We report the results in Table 3. After pretraining on all available, we provide ablation studies between training only on the prediction layer or both the prediction layer and the language embedding and between mapping to the throughput's raw values or mapping to a dictionary of labels (label2id) that can greatly shorten the range of possible values. We evaluate our results on three metrics, Spearman correlation (rank correlation), Pearson correlation (linear correlation), and percentage of accurate predictions within $\pm$ 25% of margin of error.

While statically, the transformer model performs worse or matching to the original Ithemal model [22], it performs better in another unique way. The majority of the BHive [6] data points fall under value between 20.0 and 1000.0, but the maximum can go up to 1,600,450, and the model frequently treats them as outliers. While both Ithemal and transformer struggle with large values, we observe that Ithemal can be more exact for the small data points but is really far off for these big outliers, and transformer model seems to model the big data points better but be less exact for all data points. Examples of such a difference is shown in Table 4.

(a) Proj. 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

(b) Proj. 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

(c) Reproduced Ithemal

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

Table 4: **Examples of throughput prediction made by the transformer model and Ithemal.** The left table shows the examples of the transformer model fine-tuning only the projection layer, with label2id dictionary. The middle table shows the examples of transformer fine-tuning on both projection layer and embedding space, with label2id dictionary. The right table illustrates the results of the reproduction of the original Ithemal model.

5 Discussion

This project serves as the first attempt of transformer models to be applied to low-level programming languages and opens the literature for future work to use transformer models for automatic compiler optimization tasks. While our model finds success training on the AnghaBench dataset, whether such a dataset contains a certain bias is unclear. We tested our model on a section of the AnghaBench dataset, so future work to evaluate the same model using other sources of C programs might shed light on whether the C to LLVM-IR model we built can generalize to the entire LLVM-IR language.

For translations between high-level languages, transformer models can often find exact matches of keywords on a token-to-token level and are syntactically similar to each other. However, C has a lot more abstraction than LLVM-IR, and LLVM-IR often has to represent one line of C code in multiple lines. The model can be overwhelmed by the quantity of rather unimportant lines to pinpoint the informative lines. Especially when AnghaBench contains mostly short functions, which facilitates the model's training, future work should especially examine whether the machine learning model can generate long, complicated functions with multiple branches. Attempts to directly apply long, complicated functions through Csmith did not seem to work out well.

We did not need to worry about library dependencies in the AnghaBench dataset, but in programs that do, such function definitions will show in LLVM-IR but not in C. It would add bias to library dependencies over the body of the functions themselves.

Another area for possible future work is to evaluate the translation of C to LLVM-IR without the use of BPE. While BPE can help to limit the vocab, the vocab of LLVM-IR is already limited, and its only English-based components are in strings and function names. Similar to the DynamoRIO tokenizer, a tokenizer with a fixed vocabulary might make more sense for LLVM-IR.

While the transformation of C to LLVM-IR can already be achieved with rule-based compilers, the reverse, converting LLVM-IR to humanly readable C, lacks implementations. Julia Computing has "resurrected" the LLVM C backend (`llvm-cbe`)[11], it generates generate C++ API calls to recreate the LLVM-IR basic blocks instead of recovering the control flow. Although our primitive explorations remain unsuccessful, future work on the transformer models can shed light on bettering such a reserved transformation.

We currently parse the language on a token level and use these static source-code features to characterize the language. Meanwhile, Park et al. [26] has shown success in using a control flow graph to represent the programs in their supervised machine learning model. Following a similar logic, graph neural networks [36] are an interesting framework for compiler optimization that is worth looking into.

[11] https://github.com/JuliaComputingOSS/llvm-cbe

6 Conclusion

In this paper, following successful efforts of applying transformer models on natural languages and programming languages, we explore the effectiveness of transformer on low-level compiler programs, specifically LLVM-IR and x86_64 basic blocks. Our study shows that such an unsupervised approach to low-level programs holds water and can successfully translate C to LLVM-IR while matching the state of the art for estimating basic blocks' throughput. Selecting the proper dataset and modifying the tokenization of the low-level languages can improve the performance of the model, but some constraints to the performance still remain and need further exploration.

References

[1] Toufique Ahmed and Premkumar Devanbu. Multilingual training for software engineering. *arXiv preprint arXiv:2112.02043*, 2021.

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys*, 51(5):1–42, Jan 2019. ISSN 1557-7341. doi: 10.1145/3197978. URL `http://dx.doi.org/10.1145/3197978`.

[3] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13(2), June 2016. ISSN 1544-3566. doi: 10.1145/2928270. URL `https://doi.org/10.1145/2928270`.

[4] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 133–144, 2012.

[5] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F.P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197, 2007. doi: 10.1109/CGO.2007.32.

[6] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sỳkora, Saman Amarasinghe, and Michael Carbin. Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 167–177. IEEE, 2019.

[7] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. 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, 2021.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39 (3):365–376, June 2011. ISSN 0163-5964. doi: 10.1145/2024723.2000108. URL `https://doi.org/10.1145/2024723.2000108`.

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

[11] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[12] Kaden Griffith and Jugal Kalita. Solving arithmetic word problems automatically using transformer and unambiguous representations. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 526–532. IEEE, 2019.

[13] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2020.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9: 1735–80, 12 1997. doi: 10.1162/neco.1997.9.8.1735.

[15] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning, 2020.

[16] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020.

[17] Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *SIGPLAN Not.*, 47(10):147–162, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384628. URL `https://doi.org/10.1145/2398857.2384628`.

[18] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining, 2019.

[19] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & neural unsupervised machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[20] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. ISBN 0769521029.

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

[22] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*, pages 4505–4515. PMLR, 2019.

[23] Massinissa Merouani, Mohamed-Hicham Leghettas, Riyadh Baghdadi, Taha Arbaoui, and Karima Benatchba. *A Deep Learning Based Cost Model for Automatic Code Optimization in Tiramisu*. PhD thesis, 10 2020.

[24] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1558603204.

[25] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL `https://doi.org/10.3115/1073083.1073135`.

[26] Eunjung Park, John Cavazos, and Marco A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 196–206, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312066. doi: 10.1145/2259016.2259042. URL `https://doi.org/10.1145/2259016.2259042`.

[27] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Anibal, Alec Peltekian, and Yanfang Ye. Cotext: Multi-task learning with code-text transformer. *arXiv preprint arXiv:2105.08645*, 2021.

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

[29] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*, 2019.

[30] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33, 2020.

[31] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages, 2021.

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

[33] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://www.aclweb.org/anthology/P16-1162.

[34] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, page 1–12, USA, 2013. IEEE Computer Society. ISBN 9781467355247. doi: 10.1109/CGO.2013.6495004. URL https://doi.org/10.1109/CGO.2013.6495004.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[36] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[37] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL https://doi.org/10.1145/1993316.1993532.

[38] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/dc6a7e655d7e5840e66733e9ee67cc69-Paper.pdf.