Understanding High-Level Properties of Low-Level Programs Through Transformers

Student: Zifan (Carl) Guo
MIT PRIMES
Cambridge, MA, USA
carlguo@mit.edu

Mentor: William S. Moses
MIT CSAIL
Cambridge, MA, USA
wmoses@mit.edu

***Abstract*—Transformer models have enabled breakthroughs in the field of natural language processing largely because unlike other models, Transformers can be trained on a large corpus of unlabeled data. One can then perform fine-tuning on the model to fit a specific 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. Our model shows a 49.57% verbatim match when performed on the AnghaBench dataset and a high BLEU score of 87.68. We also present another case study that analyzes x86_64 basic blocks for estimating their throughput and match the state of the art. We show through ablation studies that a collection of preprocessing simplifications of the low-level programs especially improves the model's ability to generate low level programs and discuss data selection, network architecture, as well as limitations to the use of Transformers on low-level programs.**

***Index Terms*—machine learning, NLP, compilers, LLVM, machine translation**

I. Introduction

In recent years, natural language processing has witnessed many breakthroughs due to the emergence of the Transformer machine learning model [55], pretraining objectives [14][58][33], and usage of an increasing amount of data and parameters [5]. Researchers have also started applying Transformers to other logic tasks, such as solving math problems [12][31] or understanding and performing downstream tasks on programming languages [16][45][34]. Most recently, Chen et al. [9] developed Codex, an engine that powers GitHub Copilot[1] that interprets commands in natural language and generates corresponding programs, and shows the tremendous real-world impact of language models applied on code in terms of increasing developer's efficiency.

[1]https://github.com/features/copilot

```
void foo();
void caller() {
  foo()
  foo()
}
```

```
declare void @foo()

define void @caller() {
entry:
  call void @foo() #2
  call void @foo() #3
  ret void
}
#2 = { readonly }
#3 = { writeonly }
```

Fig. 1: A sample C program (top) and its corresponding LLVM IR representation (bottom).

While the field found success in learning high-level programs, interpreting low-level programs requires additional thought. Despite being easily comprehensible because of their numerous English-based keywords, high-level programs need to be transformed through a compiler into low-level programs for the computer to understand the commands waiting to be executed. As a result, low-level programs are more verbose and less readable but more robust and precise than high-level languages. As shown in Fig. 1, there is more than twice the number of tokens in the LLVM low-level program than in the corresponding C program. Given the assumption that altering any token would cause the program to not compile, this additional verbosity provides more locations that a language model could predict incorrectly and, thus, produce un-compilable programs. Meanwhile, low-level programs tend to be more precise with more information than high-level programs; in Fig. 1, the keywords `#2` and `#3` gives the `foo()` functions attributes like `readonly` and `writeonly` that were previously hidden in the high-level programs. Due to its more verbose nature, each token in a low-level LLVM program contains less information, which makes it harder for language models to comprehend.

Allowing language models to comprehend low-level programs is essential to performance boost and optimization. Consider the code snippet in Fig. 2 that normalizes a vector.

```
__attributes__((const));
double mag(int n, const double *A);
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;
    }
}
```

Fig. 2: The left shows the original program to normalize a vector that operates in $\Theta(n^2)$ time, and the right shows the program that computes in $\Theta(n)$ time after the compiler performs *loop invariant code motion (LICM)* [38] on it.

The *loop invariant code motion (LICM)* [38] optimization pass can reduce its computation time from $\Theta(n^2)$ to $\Theta(n)$ by moving the `mag` function out of the loop. Selecting the right optimization pass can significantly improve the program's performance. Making such a selection needs comprehension on the low level because it provides extra information hidden on the high level but crucial for making the right decision. In Fig. 2, LICM is only legal if `mag` is marked `readonly` and the two pointers are marked `restrict`, meaning the two memory locations `in` and `out` don't overlap. Such information is unlikely to be explicitly written on most high level programs, whereas the low level not only contains this information when relevant but can also derive these properties.

Understanding low-level programs is important because better code optimization can greatly reduce resources devoted to running programs and, thus, reduce the operational cost of large data center applications. Moreover, such optimization can create a significant difference as Merouani et al. [37] pointed out that an optimized implementation of a deep learning neural network, such as XLNet [58], is $1.8\times$ faster than the conventional PyTorch implemented counterpart.

This need for optimization and less time-consuming, better-performing compilers is especially true with complex, large programs. The larger the program, the lower the possibility that computer scientists could manually tune the program for better efficiency. Moreover, such optimization from manual tuning is specific to the target architecture, which posts a barrier in generalizing performance to all target architectures. This obstacle, in turn, fuels the development of automatic compiler optimization that utilizes machine learning.

With the renaissance of machine learning in other fields, ML-based compiler optimization is similarly starting to be explored in depth. While most studies employ supervised machine learning strategies, more recent works have employed unsupervised techniques, such as reinforcement learning or deep learning [59][23][19]. The usage of Transformer models is growing but remains rare [51].

We attempt to tackle the compiler optimization problem by leveraging recent breakthroughs in Transformer models. We treat low-level languages, specifically, LLVM intermediate representation (IR) and x86_64 assembly, syntactic token by token, utilizing mature tokenization and preprocessing techniques that were proven successful in high-level programming languages. We built pre-trained models using the Masked Language Modelling objective [14] and fine-tuned them on parallel corpora of C to unoptimized LLVM-IR datasets, receiving proof 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 shed light on the possibility of applying Transformer architecture to provide better optimization than standard optimization flags. Our work shows that selecting 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 notations can help the model to perform better in translation. In another case study, we attempt to renovate Mendis et al. [36] 's hierarchical LSTM model that estimates throughputs of x86_64 basic blocks with Transformer models and yields results that match state-of-the-art.

II. Related Work

A. Unsupervised Language Models

Vaswani et al. [55] developed the Transformer model that revolutionizes the traditional Recurrent Neural Network (RNN) model that is slow to train and suffers from the vanishing gradients problem in long data sequences. Attempts such as LSTM [22] try to solve vanishing gradients but are slower because the process is still strictly sequential. Transformer model trains sequences in parallel through calculating attention [4], which, most importantly, can be done while disregarding their distances in the input or output sequence.

BERT establishes Masked Language Modeling (MLM) as an effective pre-train objective. Cross-lingual Language Pre-training Model (XLM) [29] upgrades the BERT model to better perform translation between multiple languages with training objectives like Back-Translation, first established by Lample et al. [30]. It also adds the Byte-Pair Encoding (BPE) [48] 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 [30].

B. Unsupervised Language Models on Programs

Researchers also applied Transformer models to high-level programming languages. Kanade et al. [27] developed a BERT model to obtain contextual embedding of Python source code, training with five classification tasks. Feng et al. [16] 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. [45] '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[2]. Roziere et al. [47] updates TransCoder with more parallel training data by taking an already trained TransCoder to generate predicted translation and leveraging an automated unit-test tool to filter out invalid prediction. Ahmed and Devanbu [2] highlighted that different implementations of the same code in multiple programming languages could preserve identifiers naming patterns well, which can serve as an anchor point for training and amplifying performance.

The amount of research on Transformer's applications on programs grows in the meantime. Researchers experimented with different implementations of the traditionally successful nature language model T5 [43] on code [41][56][11]. Novel, code-specific, pretraining objectives, such as de-obfuscation of variable names [46] and contrastive code representation[25], also appeared. Code-specific benchmark datasets, such as CodeNet [42], CodeSearchNet[24], and CodeXGLUE [34], and evaluation metrics, such as CodeBLEU[44] and APPS [21] that measures functional correctness, also were established to facilitate research in the area better. Recent Transformer research has many applications, including generating unit tests [54], AlphaCode [32], a Transformer model that solves competitive programming questions, and OpenAI Codex[9], which provides accurate suggestions to complete functions based on docstrings. Tufano et al. [53] and Drain et al. [15] attempted using Transformers to fix bugs by translating buggy code to correct ones.

C. Automatic Compiler Optimization

Ashouri et al. [3] identified two main issues for better optimization: optimization selection and phase-ordering. The former focuses on what content of optimization to adopt, and 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`"[3]. On the other hand, having a particular transformation before another might lead to better performance. The optimization problem lies in choosing a sequence of the right code transformations from the repertoire in the right order.

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

Existing compilers contain numerous preconstructed code transformation passes; for example, `Clang` 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 leaves little room for the already complicated heuristics to improve, hence, the need to apply machine learning.

The automatic compilation optimization literature contains many attempts to use supervised machine learning. For supervised machine learning, Ashouri et al. [3] split the related work based on how the model characterizes the programs. Some previous attempts select specific static features of the source code or the compilation process as proxies for the whole program 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[17]'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. [8] 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. For example, Park et al. [40] 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 [3]. However, such characterization can still be limited, as they only select parts of a program to feed into the machine learning model for training.

Many 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 [52] and the phrase ordering problem [28].

Recently, reinforcement learning and deep learning have been applied as well. With a relatively simpler objective of speedup prediction, Merouani et al. [37] applied deep learning to build a cost model implemented in the Tirasimu compiler, trying to tackle phase-ordering. Merouani et al. [37] addresses two problems in previous cost models: it only applies to basic assembly blocks instead of full programs and is heavily engineered. However, the model only assesses a few code transformations revolving around loops, such as loop fusion or tiling. Adams et al. [1] built a similar cost model to

automatically schedule Halide programs for image processing. Mendis et al. [36] developed an effective hierarchical LSTM model, Ithemal, for a similarly narrow question, estimation of throughput given x86_64 assembly basic blocks.

Zhu et al. [59] developed a framework for integrating ML into an industrial compiler and implemented inlining-for-size heuristics and register-allocation reinforcement learning models that outperform current heuristics to determine whether to adopt certain optimization passes. Jayatilaka et al. [26] focused on automatically ordering between -01, -02, and -03 pipeline based on code structure with ML. Tackling the phase-ordering problem, Huang et al. [23] optimized the phase ordering for HLS compilers with deep reinforcement learning, and Mammadli et al. [35] similarly did so with deep reinforcement learning but relied on the statically-attainable intermediate representation of the source code.

Another way to optimize programs without needing to worry about specific optimization passes is *super-optimization*, which finds a semantically equivalent but more optimized version of a given program. While doing so relied on brute force search for a long time, Bunel et al. [7] used reinforcement learning to optimize a stochastic search that improves super-optimization's efficiency. Shi et al. [49] attempted to learn symbolic expression simplification useful to super-optimization with reinforcement learning but has the possibility to contain redundancies. As the state of art, Shypula et al. [50] uses a seq2seq Transformer model for super-optimization with a Self Imitation Learning for Optimization (SILO) approach.

Despite more and more attempts of using unsupervised ML to inform the optimization process, the use of Transformer remains sparse, and most of the current attempts 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 toward utilizing Transformer models to tackle optimization selection and phase-ordering problems.

III. Model

To translate C to unoptimized LLVM-IR, we base our model structure and code implementation on TransCoder [45]: 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 [29] for cross-lingual natural language translation: initialization, language modeling, and back-translation. We use the first two steps but adopt a 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.

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

A. Preprocessing

To process into the ML pipeline, we use separate tokenizers for C and LLVM-IR similar to Roziere et al. [45] 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 preprocessor directives like `#define`, Clang would tokenize it into two tokens # and `define` respectively, while Tree-sitter keeps it as one token. We use the Clang C tokenizer because of its internal logic's similarity to the LLVM-IR tokenizer that we implement, which utilizes similar libraries. We extend 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.

B. Training Objectives

Lample et al. [30] concluded the importance of pretraining in unsupervised machine translation by mapping similar sequences with similar meanings, regardless of the languages. Roziere et al. [45] 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 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 [45]. We show that enough anchor points exist 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 [14] following Roziere et al. [45]. 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 encoder matches the architecture of the pretrained XLM model, the decoder needs extra parameters on the source attentions, randomly initialized following Lample and Conneau [29]. As the decoder has never been 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

[4]https://tree-sitter.github.io/tree-sitter/
[5]https://github.com/facebookresearch/CodeGen
[6]https://github.com/pybind/pybind11
[7]https://github.com/glample/fastBPE

on a corrupted version with additional noise, first established in Lample et al. [30]. The noise is randomly masking, removing, and shuffling tokens in the input sequences. This step trains the encoder to be robust against noise to perform the latter machine translation objective better [45].

With the pretraining MLM and denoising auto-encoding objectives, the model would be able to translate but has low performance because it depends on the inherent and unchangeable "cross-linguality" based on the number of common anchor points [45]. Fine-tune tasks are adopted to boost the model's performance.

TransCoder [45] and XLM [29] are trained on the back-translation objective, which translates the sequence in the source language to the target language and back to the source language, on which the loss function is performed. However, as TransCoder-ST [47] identifies, back-translation is a mediocre solution to the lack of parallel data using only monolingual data. Back-translation is less direct than machine translation and creates more noise. In the case of translating from C to LLVM-IR, we can easily access such parallel corpus as long as the C program can compile and choose to fine-tune with a machine translation task.

We train machine translation and denoising auto-encoding in parallel until they converge.

C. Preprocessing Modifications

We first clean up the C data before compiling to generate LLVM-IR with `clang -E`, which writes out all the preprocessing directives such as imported libraries. 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`, `store`, or `getelementptr inbounds`, the data type always appears twice, once as itself and another as the pointer to it. In this case, we remove one of the two appearances and construct a detokenizer that can restore it.

We remove all comments as they are filtered out in the compilation process 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`. Of course, the string information would be lost when we only extract functions to train. 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 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, i32 }}` is even harder to comprehend. Griffith and Kalita [18] 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 `type`s in prefix notation, converting the above struct into `STRUCT 5 ARR 3 4 x i8 i32 STRUCT 2 i8 i32`. By recording the length of the `struct`, the detokenizer can faithfully restore them to evaluate the model's performance. Representing data in prefix notation is easier for the Transformer model to understand.

IV. Experiment

A. Training Details

Basing off of Roziere et al. [45]'s TransCoder, we train our model with a transformer of 6 layers, 8 attention heads, with a single encoder and a single decoder for both high-level and low-level programming languages. At training time, we use batches of around 3500 tokens. We use the GELU [20] as an activation function. We add in a 10% dropout rate and a 10% attention dropout rate. We optimize the model with the Adam optimizer and a learning rate of 10^{-4}. It is worth noting that we have fewer computing resources than the researchers producing the work to which this paper is referencing; we train using 1 GeForce RTX 3090 GPU while Roziere et al. [45] trained with 32 V100 GPUs. 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.

B. Training Data

We have considered multiple data sources for our training data, including CSmith [57], Project CodeNet [42], GitHub

Google BigQuery[8], and AnghaBench [13].

CSmith by Yang et al. [57] 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 do 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.

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 of 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.

[8]https://console.cloud.google.com/marketplace/details/github/github-repos
[9]https://onlinejudge.u-aizu.ac.jp/home
[10]https://atcoder.jp/

C. Evaluation

We evaluate our results on four metrics, the training accuracy generated by the loss function, perfect reference matches, the industry convention BLEU [39] score, and compilation accuracy.

Training accuracy describes how well the model performs on the machine translation objective at hand with the training data. The rest considers a new batch of testing data that the model has never seen before. Reference match refers to a percentage out of all translation prediction units that match the ground truth verbatim. The BLEU [39] score, on the other hand, is a widely accepted evaluation metric for natural language translation that evaluates the quality of the text of predicted translation by comparing its similarity with their referencing ground truth. The BLEU score performs such evaluation by taking the geometric mean of multiple modified *n-gram* (unigram, bigram, trigram, and 4-gram) precision scores, with 0 as completely different and 100 as exactly the same. The BLEU score presented in the paper is an average of the BLEU scores performed on each translation case. Lastly, for the datasets for which we successfully constructed a detokenizer, we report the compilation accuracy, namely the number of programs out of the total number that were successfully compiled. Because those that match the ground truth verbatim would definitely compile, the compilation accuracy is always higher than the reference match score. For high-level programs, prior research has concluded that functional correctness is the best evaluation metric for such machine learning models [45][9][47], and since we can only compile, but not run, a low-level program, compilation accuracy is the best proxy.

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. Despite BLEU's ease to use, it has its limitation that it does not evaluate naturalness unique to programming languages, such as important syntactic and semantic features, which are especially critical for interpreting code with such rigid structures.

Recent development such as CodeBLEU [44][34] offers a new evaluation metric that updates BLEU to be code-specific, taking an additional AST (Abstract Syntax Tree) and a dataflow graph comparison into consideration to evaluate the code's structure. However, CodeBLEU requires language-by-language specific implementation and lacks portability. As Ren et al. [44] aims CodeBLEU to serve the community of machine learning on high-level programs, it is yet to be applied to evaluating low-level programs like LLVM-IR. Building a metric to evaluate low-level programs is of future interest.

TABLE I: **Results of unsupervised machine translation on the AnghaBench test set.** Ablation studies with various preprocessing modifications. We apply syntactic cleaning and representation in prefix notation based on the original dataset, documented in the second column labeled C̈leaned,änd train another model with an additional restoration of global variables in the third column. The fourth column trains LLVM data with the `-O1` flag.

AnghaBench Dataset	Original	Cleaned	Cleaned & Global	-O1
Training Acc.	99.03	99.60	99.36	97.87
Reference Match	13.33	**49.57**	38.61	38.73
BLEU	69.21	**87.68**	82.55	77.03
Compilation Acc.	14.97	NA	41.45	NA

TABLE II: **Results of unsupervised machine translation on the Csmith and CodeNet test set.** The models trained on both datasets are subpar to that on the AnghaBench dataset mostly due to a limited amount of and their inherent unnaturalness.

	Csmith	CodeNet
Testing Accuracy	90.73	93.66
Reference Match	N/A	5.76
BLEU	43.39	51.01

D. Results

The current results are reported in the following tables. We report the results on our AnghaBench test set, with ablation studies with various preprocessing modifications, in Table I. In the table, the first column labeled *Original*, we show the results after training the model on the original, unmodified dataset on which we only perform the standard `clang -E` preprocessing to rid the preprocessing directives. Despite giving us a high training accuracy of 99.03%, the model does not generalize well to unseen testing data, giving us the lowest reference match accuracy (13.33%) and BLEU score (69.21); such low performance is most likely due to the sheer amount of uninformative tokens that overwhelm the number of informative tokens. The second column, labeled *Cleaned*, illustrates the training result after converting data representation to prefix notations. As we previously explained in the Section III , prefix notations remove uninformative tokens such as brackets and commas and condense such information into one token that describes the `array` or `struct`'s length. Such an endeavor boosts the performance by a large amount, but doing so ignores definitions of global variables. Global `struct` definitions are permenantly lost on the function level, which, in turn, prevent us from completely detokenizing the programs and compiling them as an evaluation metric.

```
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
 }
```

Fig. 3: **Example of LLVM-IR prediction with our Transformer model.** The top is the original source code in C, and the bottom is a verbatim copy in LLVM-IR to the excepted compiler output.

In the third column, labeled *Cleaned & Global*we report the results of the model on AnghaBench dataset after converting data structures in infix notation to prefix notation and writing out global variables and structs as their respective declarations and definitions. While doing so makes sure we can later restore the global definitions to compile the programs, it expands the already complex syntax and assumes information that does not exist on th high level, making it harder for language models to understand. Our result justifies this claim by showing that this preprocessing modification slightly hinders the model's performance, in comparison to only cleaning the syntax and writing in prefix notation. The fourth column shows the result of training on LLVM-IR optimized with `-O1` flag. It serves as a preliminary look into the possibility of language models understanding both optimized and unoptimized low-level programs and shows promising results.

We give an example of such unsupervised translation from C to LLVM-IR tested on AnghaBench in Fig. 3. We conclude that applying prefix notation transformation and removing redundant language syntax help the model to perform better, but writing out global variable and struct definitions inside functions, despite being necessary for accurate recovery, hinders the performance.

We report the results of training on Csmith and CodeNet data in Table II. We observe that the transformer model performs better on the AnghaBench dataset than on Csmith [57] and CodeNet[42], giving better reference matches and BLEU sscores AnghaBench dataset is more expansive than both Csmith and CodeNet, and a lot closer as a proxy to the humanly written high-level programs than Csmith. While we find moderate success in training with CodeNet data, we worry that a model solely based on CodeNet contains internal biases and cannot generalize well to the LLVM-IR language because it contains duplications of similar answers.

V. Throughput Estimation of x86_64 Assembly Basic Block

In another separate case study, we have also attempted to renovate Mendis et al. [36]'s Ithemal, which utilizes a hierarchical LSTM model, with transformer models. Accurate throughput estimation is an essential tool to inform the computer on how to choose the proper optimization passes.

A. Setup

We train our transformer model on the BHive [10] 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 Fig. 4. We

TABLE III: **Results of applying transformer model to estimate the throughput of x86_64 basic blocks.** Spearman correlation measures the strength and direction of monotonic association between the given assembly program and its throughput, while Pearson correlation evaluates only the linear relationship between the two variables. For validation accuracy, as long as the data point is within a 25% margin of error, it counts as correct.

	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 IV: **Examples of throughput prediction made by the Transformer model and Ithemal.** While Ithemal produces more accurate results for smaller data points, it struggles with large data points and can produce results drastically different from the ground truth; such data points are marked as red in the table.

(a) 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

(b) 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

(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

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

110.0

Fig. 4: The above figure is an example of the data points in the BHive dataset. The top illustrates one x86_64 basic block, and the bottom shows its corresponding throughput as a numerical value.

follow the preprocessing structure outlined in Ithemal, adopting a DynamoRIO [6] 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, 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.

B. Results

We pre-train 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 III. The transformer model yields results that match the original Ithemal model. 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. Among the different ablations of transformer models, fine-tuning with the 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%.

While statically, the transformer model performs worse or matching to the original Ithemal model [36], it performs better in another unique way. The majority of the BHive [10] data points fall under a 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 the Transformer model seems to model the big data points better but be less exact for all data points. Examples of such a difference are shown in Table IV.

VI. Discussion

This project serves as the first attempt of transformer models to be applied to low-level programming languages. It opens the literature for future work to use transformer models for automatic compiler optimization tasks. Through our two case studies, we believe that a cross-lingual model on both high-level and low-level programs can perform better, as the two

levels complement each other to build a holistic understanding of the program at hand.

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. However, C has much 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 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 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.

Another area for possible future work is to evaluate the translation of C to LLVM-IR without using 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 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.

VII. Conclusion

In this paper, following successful efforts of applying transformer models to natural languages and programming languages, we explore the effectiveness of the Transformer model 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 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 model's performance, but some constraints to the performance remain and need further exploration.

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

Acknowledgment

We thank Susan Tan, Yebin Chon, and Johannes Doerfert for thoughtful discussions on using similar machine learning models for decompilization from LLVM-IR to C, the real-world application of the de-obfuscation objective [46]pre-train objective, and its implementation on the C language. We thank MIT and the MIT PRIMES program for making the research possible. This research was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323; in part by LANL grant 531711; and in by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4): 1–12, 2019.

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

[3] 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.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[6] 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.

[7] Rudy Bunel, Alban Desmaison, M Pawan Kumar, Philip HS Torr, and Pushmeet Kohli. Learning to superoptimize programs. *arXiv preprint arXiv:1611.01787*, 2016.

[8] 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.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] 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.

[11] Colin B Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers. *arXiv preprint arXiv:2010.03150*, 2020.

[12] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[13] 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.

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

[15] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–8, 2021.

[16] 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.

[17] 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.

[18] 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.

[19] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. *NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning*, page 242–255. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450370479. URL https://doi.org/10.1145/3368826.3377928.

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

[21] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

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

[23] 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.

[24] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[25] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

[26] Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, EunJung Park, and Johannes Doerfert. *Towards Compile-Time-Reducing Compiler Optimization Selection via Machine Learning*. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450384414. URL https://doi.org/10.1145/3458744.3473355.

[27] 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.

[28] 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.

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

[30] 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.

[31] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems

with language models. *arXiv preprint arXiv:2206.14858*, 2022.

[32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[33] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL https://arxiv.org/abs/1907.11692.

[34] 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.

[35] Rahim Mammadli, Ali Jannesari, and Felix Wolf. Static neural compiler optimization via deep reinforcement learning, 2020. URL https://arxiv.org/abs/2008.08951.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

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

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

[47] 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.

[48] 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.

[49] Hui Shi, Yang Zhang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. Deep symbolic superoptimization without human knowledge. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=r1egIyBFPS.

[50] Alex Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. Learning to superoptimize real-world programs. *arXiv preprint arXiv:2109.13498*, 2021.

[51] Alex Shypula, Pengcheng Yin, Jeremy Lacomis, Claire Le Goues, Edward Schwartz, and Graham Neubig. Learning to superoptimize real-world programs. *arXiv preprint arXiv:2109.13498*, 2021.

[52] 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.

[53] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions*

on Software Engineering and Methodology (TOSEM), 28 (4):1–29, 2019.

[54] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

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

[56] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[57] 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.

[58] 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.

[59] Hang Zhu, Varun Gupta, Satyajeet Singh Ahuja, Yuandong Tian, Ying Zhang, and Xin Jin. Network planning with deep reinforcement learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 258–271, 2021.