Enabling Transformers to Understand Low-Level Programs

Zifan (Carl) Guo
MIT PRIMES
carlguo@mit.edu

William S. Moses
MIT CSAIL
wmoses@mit.edu

2022 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-9786-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/HPEC55821.2022.9926313

***Abstract*—Unlike prior approaches to machine learning, Transformer models can first be trained on a large corpus of unlabeled data with a generic objective and then on a smaller task-specific dataset. This versatility has led to both larger models and datasets. Consequently, Transformers have led to breakthroughs in the field of natural language processing. Generic program optimization presently operates on low-level programs such as LLVM. Unlike the high-level languages (e.g. C, Python, Java), which have seen initial success in machine-learning analyses, lower-level languages tend to be more verbose and repetitive to precisely specify program behavior, provide more details about microarchitecture, and derive properties necessary for optimization, all of which makes it difficult for machine learning.**

In this work, we apply transfer learning to low-level (LLVM) programs and study how low-level programs can be made more amenable to Transformer models through various techniques, including preprocessing, infix/prefix operators, and information deduplication. We evaluate the effectiveness of these techniques through a series of ablation studies on the task of translating C to both unoptimized (`-O0`) and optimized (`-O1`) LLVM IR. On the AnghaBench dataset, our model achieves a 49.57% verbatim match and BLEU score of 87.68 against Clang `-O0` and 38.73% verbatim match and BLEU score of 77.03 against Clang `-O1`.

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

I. Introduction

In recent years, natural language processing has experienced a variety of breakthroughs due to the emergence of the Transformer machine learning model [49], pretraining objectives [11, 28, 52], and the usage of an increasing amount of data and parameters [4]. Researchers have also started applying Transformers to other logic tasks like solving math problems [9, 26] and a variety of tasks on programming languages such as machine translation [40], bug detection [12], and code generation [29]. Recently, Codex [7] was trained on a large corpus of public GitHub repositories and powers the Github Copilot application for AI programs autocompletion [1].

While Transformer models have found success in learning high-level programs, interpreting low-level programs requires additional thought. While high-level programs contain many English-based keywords to simplify the process of writing code, traditional compilers first transform programs into unambiguous low-level representations that contain sufficient information to enable optimization and analysis passes. As a result, low-level programs tend to be more verbose and less readable, but more robust and precise than high-level languages. As demonstrated in Fig. 1, the LLVM representation of simple C++ code contains roughly twice the number of tokens. By simply assuming that altering any token would cause the program not to compile, this additional verbosity provides more locations that a language model could produce an erroroneous token and conseuquently emit a program that cannot be compiled. This additional verbosity, however, allows low-level programs to specify more program semantics than high-level programs. As an example, in Fig. 1, the compiler can add attributes like `nocapture` and `readonly` to the parameters of `printf()`, specifying that the arguments to the function are not captured or written to, respectively. While this information could have been deduced from the high-level program itself, it was implicit in the definition of `printf()`, rather than explicitly marked in the program. Similarly, whereas the function calls to `printf` inside the `Derived` and `Base` class constructors are implied within C++, on the LLVM level `printf` is called explicitly.

These additional properties available in low-level programs are essential for optimization. Consider the code snippet in Fig. 2 that normalizes a vector, with the explicit program properties written on the high-level code for ease. The *loop invariant code motion (LICM)* [33] optimization pass can reduce the runtime of the `norm` function from $\Theta(n^2)$ to $\Theta(n)$ by moving the call to `mag` out of the loop so that `mag` can be computed once and reused for every iteration. In Fig. 2, the LICM optimization is only legal if `mag` is marked `readonly` and the two pointers are marked `restrict`, meaning that the memory locations `in` and `out` do not overlap. These semantics are required since otherwise either the call to `mag` or the store to `out` would be able to overwrite `in`, potentially resulting in a different value for the magnitude on each iteration. These properties tend not to be explicitly specified in high-level programs, whereas low-level representations like LLVM have the ability to represent these semantics but also include analysis passes to derive these properties automatically.

While Transformers have historically only been trained on high-level programs, an open question remains regarding their effectiveness on low-level languages. Since low-level languages like LLVM are where most optimizations and analyses are applied, successfully applying Transformers on low-level programs can open the door for automatic general-program optimization. This paper aims to answer the question of how

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

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

```c
#include <stdio.h>
class Base { public:
  Base() {
    printf("Called Base()");
}  };
class Derived : public Base {
public:
  Derived() {
    printf("Called Derived()");
  }
  int square(int x) {
    return x * x;
}  };
int f(int x) {
  return Derived().square(x);
}
```

```
@.str = private constant [17 x i8] c"Called Derived()\00"
@.str.1 = private constant [14 x i8] c"Called Base()\00"

define i32 @_Z1fi(i32 %0)  {
  %2 = call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1)
    getelementptr inbounds ([14 x i8], [14 x i8]* @.str.1, i64 0, i64 0))
  %3 = call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1)
    getelementptr inbounds ([17 x i8], [17 x i8]* @.str, i64 0, i64 0))
  %4 = mul nsw i32 %0, %0
  ret i32 %4
}

declare i32 @printf(i8* nocapture readonly, ...)
```

Fig. 1: A sample C program (left) and its corresponding LLVM IR (right). The LLVM IR is more verbose but explicitly writes out function calls and attributes that are hidden on the high level.

```c
__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);
}
```

```c
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 an unoptimized program to normalize a vector that runs in $\Theta(n^2)$ time. The right shows an optimized version of the same program that runs in $\Theta(n)$ time after performing the *loop invariant code motion (LICM)* [33] optimization.

effective such models are presently on low-level programs and what techniques can be applied to make analyzing low-level programs more effective. To answer this question, we apply Transformer models on LLVM IR, a common compiler intermediate representation used for optimization and code generation in various languages, including C, C++, Julia, Rust, Swift, Fortran, etc. Specifically, we preprocess and train a Transformer model on LLVM IR by applying best-practice techniques leveraged on high-level programming languages, as well as novel techniques specific to LLVM and low-level programs. To evaluate the overall effectiveness of the model, we focus on a case study of training a replacement for a traditional C compiler, i.e., training a model to translate C functions into unoptimized and optimized LLVM IR. Specifically, we have the following contributions:

- We implement a preprocessing pipeline for low-level programs like LLVM IR for training Transformer models.
- We describe several techniques for improving the performance of transformer models on low-level code, with ablation studies of their efficiency.
- We demonstrate end-to-end translation between C and both unoptimized (`-O0`) LLVM IR and optimized (`-O1`) LLVM IR via a Transformer model.

II. Related Work

A. Unsupervised Language Models

Transformer models [49] and subsequent extensions like BERT [11] enable training on a large corpus of unlabeled data and then fine-tuning on task-specific labeled data. Cross-lingual Language Pretraining Model (XLM) [24] further allows training of multiple languages in one model, establishes fine-tuning objectives like back-translation [25], and adds byte-pair encoding (BPE) [43] to the preprocessing process, which splits words into sub-words to condense an open-vocabulary task into a fixed vocabulary task [25]. Recent work [4, 38] shows that pretraining with more data from different languages and more model parameters can reach better results on downstream tasks after fine-tuning.

B. Unsupervised Language Models on Programs

Researchers have applied Transformers to high-level programming languages. Kanade et al. [22] and Feng et al. [13] transfer the BERT model to capture the semantic similarity between natural and programming languages and show that pretraining on code is effective. Researchers have also experimented with different implementations of the traditionally successful language model T5 [38] on code [8, 36, 50].

TransCoder [40] is an unsupervised model that translates between C++, Java, and Python, based on open-sourced GitHub monolingual source code data accessed through Google BigQuery[2]. Roziere et al. [42] update TransCoder with parallel training data by taking a pretrained model to generate predicted translation and leveraging an automated unit-test tool to filter out invalid predictions. Ahmed and Devanbu [2] highlight that the same code in multiple programming languages could preserve identifiers and naming patterns well, serving as anchor points for training and amplifying performance.

The literature on code-specific machine learning is actively growing. Recent work explores code-specific pretraining objectives like de-obfuscation of variable names [41]

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

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

and contrastive code representation [20]; code-specific benchmark datasets like CodeNet [37], CodeSearchNet [19], and CodeXGLUE [29]; and evaluation metrics like CodeBLEU [39] and APPS [17]. Recent Transformer research includes generating unit tests [48], AlphaCode [27], a Transformer model that solves competitive programming questions, and Codex [7], which provides accurate suggestions to complete functions based on docstrings. Tufano et al. [47] and Drain et al. [12] use Transformers to fix bugs by translating buggy programs to correct ones.

C. Automatic Compiler Optimization

There has been a plethora of work in the field of machine learning-assisted optimization. Whereas our work aims to explore how effectively Transformers can be used to entirely substitute for a compiler's code generation and optimization phases in their entirety, most pieces of prior work focus on applying machine learning to specific components of the compilation or optimization pipeline and tend to rely on supervised learning.

Several previous supervised learning-based approaches extract program features, such as static source code [14], performance counters, [6] or control flow graphs (CFG) [35]. There have also been reinforcement learning and deep learning tools for feature extraction, such as building cost models [1, 32] or estimating throughputs [31].

Both optimization selection and phase-ordering have also been explored through genetic algorithms [23, 46] and reinforcement learning [18, 30]. MLGO [53] updates traditional LLVM heuristics for inlining-for-size and register-allocation through reinforcement learning models. Jayatilaka et al. [21] automatically determine whether one should use the `-O1`, `-O2`, or `-O3` optimization pass sequences.

Super-optimization, or optimizing programs without considering specific optimization passes, involves finding a semantically equivalent but more optimized version of any given program. While doing so relies on brute force search, recent developments show promising results of *super-optimization* with reinforcement learning [5, 44] and seq2seq Transformer models [45].

Like this work, Armengol-Estapé and O'Boyle [3] aim to explore how effectively Transformers can replace traditional compilers. Unlike this work, Armengol-Estapé and O'Boyle [3] focuses on platform-specific x86 code-generation, whereas we target LLVM IR–a slightly higher-level program representation where many existing optimizations are performed.

III. Base Model

To translate C to unoptimized LLVM IR, we build off of TransCoder [40]: 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 [24] 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 machine translation objectives only on the standalone, static function.

A. Preprocessing

To process into the ML pipeline, we use separate tokenizers for C and LLVM IR similar to Roziere et al. [40] because different languages may use the same keywords to convey drastically different semantics. 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, when parsing the `#define` directive, Clang generates two tokens, `#` and `define`, whereas Tree-sitter keeps it as one token. We use the Clang C tokenizer because of the similarity of its internal logic to that of the LLVM IR tokenizer. Using `PyBind11`[6] we exposed the LLVM lexer (`LLLexer`) to Python, which we use as our tokenizer[7]. The LLVM tokenizer provides the type and string representation of each token, allowing us to parse the relevant information. Using fastBPE[8], we then learn BPE codes on the concatenation of these tokens and split them into subword units.

B. Training Objectives

Lample et al. [25] demonstrate the importance of pretraining in unsupervised machine translation by mapping similar sequences with similar meanings, regardless of the languages. Roziere et al. [40] identify that 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 a worse cross-lingual representation than a translation between two high-level languages because of the higher syntactical and structural differences between C and LLVM. In an analogue to NLP, an English-French model would have more "cross-linguality" than an English-Chinese model because of the similar alphabet [40] and sentence structure. 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 [11] following Roziere et al. [40]. 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

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

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

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

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

[7]https://github.com/wsmoses/llvm-tokenizer

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

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

the source attentions, randomly initialized following Lample and Conneau [24]. 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 on a corrupted version with additional noise, first established in Lample et al. [25]. The noise randomly masks, removes and shuffles tokens in the input sequences. Applying this noise to the training process makes the encoder more robust, improving its results on the machine translation objective [40].

With the pretraining MLM and denoising auto-encoding objectives, the model has a general sense of the two languages. While raining on these two tasks alone allows the model to translate some programs, its accuracy is limited by the low number of common anchor points between the LLVM and C. Therefore, to boost the model's performance, we use machine translation as a fine-tuning task. In particular, we use a language-parallel data corpus of C and LLVM IR programs.

TransCoder [40] and XLM [24] are trained on the back-translation objective, which considers the loss between the original program in C and a program in C after translating to LLVM IR and back. However, TransCoder-ST [42] identifies that back-translation creates more noise, and the machine translation objective that considers the loss between the original program and the directly translated program is preferred to produce better results. As a result, back-translation is a compromise that should only be used in the absence of a dataset with many equivalent programs in different languages. In the case of translating from C to LLVM IR, we can easily access a parallel dataset since one can directly generate LLVM IR by compiling C programs. As a result, we choose to fine-tune with a machine translation task.

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

IV. Improving the Model

We introduce several optimizations for improving the effectiveness of the model's ability to understand and generate LLVM IR from C.

a) Code Expansion & Cleaning: We perform several preprocessing steps on top of the baseline. We first clean up the C code before compiling to generate LLVM IR with `clang -E`, which expands preprocessing directives such as pasting the definition of imported libraries, compile-time constants, and more. Such expansion is necessary for our preprocessing pipeline to run properly.

We also remove specific LLVM tokens that do not significantly change the semantics of the program, thereby reducing the amount of information that the model needs to learn and thus enabling better training. We remove the data layout, target hardware architecture, alignments, global attribute groups, and metadata. Finally, we remove comments.

b) Redundancy Elimination: In some statements like `load`, `store`, or `getelementptr`, the data type appears twice, once as itself and again as a pointer. In this case, as shown in Fig. 3, we remove one of the two appearances and construct a detokenizer to restore the second occurrence.

```
%4 = load i32**, i32*** %2
```

```
%4 = load i32** %2
```

Fig. 3: A `load` statement (top) in LLVM and the same statement after removing redundant type information (bottom).

```
%struct.TYPE_8__ = type { i32, i32, i64 }
...
%21 = alloca %struct.TYPE_8__, i64 %19
```

```
%struct.TYPE_8__ = type { i32, i32, i64 }
...
%21 = alloca { i32, i32, i64 }, i64 %19
```

Fig. 4: The definition and use of a struct type before (top) and after inlining the type definition.

c) Global Name Inlining: As we want to eventually compile the generated programs but only train on individual functions rather than entire files, some information is lost and cannot be recovered. While names and types of externally-visible global variables or function declarations are immediately available from their use, the definition of any `struct` or `class` is permanently lost and would hinder the program's compilation. To remedy this problem, we inline the references of non-recursive `struct`'s, as shown in Fig. 4.

While this process adds complexity to the model, it allows one to compile the generated functions later. The performance impact of this change is discussed in Section V.

d) String Name Inlining: Similar to `struct` types, for each string constant, LLVM IR automatically generates a global variable with names such as `@.str.1` or `@.str.2`. Like other private global variables, the actual content of the string is lost when we only extract functions. One way of resolving this issue is to similarly inline the definition of the string at the location of its use. In the case of constant strings, however, we chose to leave the variable as a length-specific placeholder, which can be filled in with a valid value after translation and prior to execution.

e) Type Prefixing: The bitcode representation of LLVM types is difficult for the model to learn. In particular, arrays and structs in LLVM IR (see Fig. 5) require the Transformer to consider the scope of the array and where the `[]` ends. Building off of the work of Griffith and Kalita [15], which shows improvements of Transformer models in solving arithmetic problems specified in prefix notation instead of the conventional infix notation, we similarly decide to rewrite such structure definitions to be "prefix"-like. Specifically, we remove the structures of `[]` or `{}` and write out the types in prefix notation, as shown in Fig. 5. Furthermore, we remove the commas inside the data structures because they are unnecessary for de-tokenization. By recording the length of the `struct`, the detokenizer can faithfully restore them to evaluate the model's performance.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

```
[3 x i32] [i32 1, i32 2, i32 3]
{ [4 x i8], i32, { i8, i32 }}
```

```
ARR 3 3 x i32 ARR 6 i32 1 i32 2 i32 3
STRUCT 5 ARR 3 4 x i8 i32 STRUCT 2 i8 i32
```

Fig. 5: An LLVM IR array and struct definition in infix notation (top) and prefix notation (bottom).

V. Experiment

A. Training Details

Like TransCoder [40], 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. We use batches of around 3500 tokens and GELU [16] as our 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}. Experiments were trained on a single GeForce RTX 3090 GPU (notably fewer than the 32 V100 GPUs in the original Transcoder paper).

B. Training Data

We train our model on multiple datasets: *CSmith* [51] (a randomized test-case generation tool for C programs); *Project CodeNet* [37] (a collection of solutions submitted by the public to competitive programming websites); and *AnghaBench* [10] (a benchmark of more than 1 million compilable C functions constructed from crawling C files on GitHub).

In selecting a good training dataset, we need to simultaneously ensure that the dataset is large enough to saturate the model and that the C programs can be compiled and thus generate LLVM IR. While CSmith meets both thresholds, we saw poor results primarily due to its randomness and the lack of proximity to human written code.[9] While CodeNet programs all successfully compiled and there were many submissions, they all tended to respond to the same question. As a result, the dataset lacks diversity—leading to overfitting and poor results. The AnghaBench dataset not only has a larger amount of data but all programs are cleaned and compilable by applying type-inference to reconstruct the missing definitions (e.g. declarations of auxiliary functions, types, etc.). Moreover, having only one extracted function in each file eases training, and our model finds the most success in this dataset.

C. Evaluation

We evaluate our results on four metrics:

- **Training Accuracy** describes how well the model performs on the machine translation objective in the fine-tuning phase with the training data.
- **Reference Match** denotes what percentage of programs from a test dataset matches the ground truth verbatim when run through our model.
- **BLEU** [34] score is a common metric for natural language translation that evaluates the quality of the text of predicted translation by comparing its similarity with their referenced ground truth.[10]
- **Compilation accuracy** counts the number of programs whose translation was successfully compiled. This metric can only be applied to datasets whose lost information (see global variables in Section IV) can be recovered. As prior research on high-level programs determined that functional correctness is the best evaluation metric for machine learning models [7, 40, 42], compilation success is an apt proxy without the corresponding inputs to run all the programs on.

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 may render BLEU score uninformative as the model can translate into illegal or unintelligible LLVM IR but translate back to proper C. Since we can easily generate parallel matching data for C and LLVM IR using the baseline Clang compiler, our model's direct machine translation makes the evaluation of BLEU score more informative. Despite BLEU's ease of use, it is possible to generate programs that do not appear to be human or even compiler-generated but have a high BLEU score. As such, training to maximize a BLEU score may downplay the importance of learning language-specific syntactic and semantic features, which are especially critical for interpreting code.

Recent work like CodeBLEU [29, 39] 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 CodeBLEU aims to serve the community of machine learning on high-level programs, it is yet to be implemented to evaluating low-level programs like LLVM IR. Building a metric to evaluate low-level programs is of interest to future work.

D. Results

The results are reported in two tables. We report the results on the AnghaBench test set, with ablation studies with various preprocessing optimizations, in Table I. In the 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 yielding a high training accuracy of 99.03%, the model does not generalize well to unseen testing data, resulting in the lowest reference match accuracy (13.33%) and BLEU score (69.21). This low performance is likely due to the number of uninformative tokens that overwhelm the number of informative tokens. The *Cleaned* column illustrates

[9]Additionally, CSmith functions are quite long, whereas current machine learning models perform better on shorter sequences.

[10]The BLEU score performs evaluation by taking the geometric mean of multiple modified *n-gram* (unigram, bigram, trigram, and 4-gram) precision scores, with 0 representing completely different values and 100 as the same values. We take the average BLEU score on relevant inputs.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

TABLE I: **Results of unsupervised machine translation on the AnghaBench test set.** We enable various preprocessing optimizations described in Section IV. We train on the unmodified dataset (Original), with syntactic cleaning (Cleaned), with prefix notation (Prefix), with a restoration of global variables (Prefix & Global), and while targeting of optimized LLVM (`-O1`).

AnghaBench	Original	Cleaned	Prefix	Prefix & Global	-O1
Training Acc.	99.03	97.84	99.60	99.36	97.87
Reference Match	13.33	21.15	**49.57**	38.61	38.73
BLEU	69.21	72.48	**87.68**	82.55	77.03
Compilation Acc.	14.97	N/A	N/A	43.07	N/A

TABLE II: **Results of unsupervised machine translation on the Csmith and CodeNet datasets.** The models trained on these datasets are subpar to the AnghaBench dataset primarily due their size and unnatural inputs.

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

```
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. 6: **Example of LLVM IR prediction with our Transformer model.** The top is the original source code in C, and the bottom is the expected LLVM IR (for which a precise match is generated by the model).

the training result after we deduplicate information, which performs slightly better than the original dataset. We report in the *Prefix* column the training result after deduplication and converting data representation to prefix notation. Removing the additional brackets, commas, and additional context needed to parse LLVM types significantly improves performance but ignores definitions of global variables. Global `struct` definitions are permanently lost on the function level, preventing us from detokenizing and subsequently compiling the programs for evaluation.

The *Prefix & Global* column reports the model's results on the 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 expanding globals ensures detokenization and compilation can occur, it adds complications to the program, making it harder for the model to understand. This rationale is demonstrated in the data, with slightly worse results than *Prefix* alone. The *-O1* column shows the result of training on LLVM IR optimized with `-O1` flag and serves as an initial study of the possibility of language models understanding both optimized and unoptimized low-level programs, demonstrating promising results. An example of machine-learned translation from C to unoptimized LLVM IR (trained on AnghaBench) is shown in Fig. 6.

The results of training on Csmith and CodeNet data are shown in Table II. We observe that the Transformer model performs better on the AnghaBench dataset than on Csmith [51] and CodeNet [37], giving better reference matches and BLEU scores as the AnghaBench dataset is more expansive than both Csmith and CodeNet, and a better proxy for humanly written code than Csmith. While a model trained on CodeNet data is moderately successful, it also contains internal biases and cannot generalize well to the LLVM IR language due to containing many similar programs.

VI. Conclusion & Future Work

In this paper, following successful efforts of applying Transformer models to both natural and high-level programming languages, we explore the effectiveness of the Transformer model on low-level programs. Specifically, we explore the effectiveness of transformers on LLVM IR, the typical compiler intermediate language used for optimizations by several programming languages. Specifically, we perform a case study exploring how effectively a transformer can act as a replacement for the code generation and optimization pipelines of a traditional C compiler by automatically translating C to (unoptimized & optimized) LLVM. While the results would certainly not encourage anyone to immediately replace their existing compilation framework with a neural network, they are nevertheless promising. Specifically, they demonstrate that, unlike the existing ML-based optimization approaches that rely on injecting machine learning to specific heuristics within a broader compilation pipeline, whole-program analysis and optimization with machine learning will have a future as the models, techniques, and datasets mature. Moreover, we also demonstrate how existing Transformer models can be better applied to low-level programs through the use of LLVM-specific preprocessing optimizations.

There are several avenues for potential future work. For example, in fine-tuning the Transformer model structure for low-level programs, further study could explore training a monolingual model solely on LLVM IR, de-compiling LLVM IR to humanly readable C, and preprocessing LLVM IR for Transformers without applying BPE [43] as the vocabulary of LLVM IR is already limited.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

ACKNOWLEDGMENT

The authors would like to thank Susan Tan (Princeton), Yebin Chon (Princeton), and Johannes Doerfert (ANL) for thoughtful discussions on using machine learning within LLVM, including representations of LLVM IR, decompilation from LLVM-IR to C, the real-world application of the deobfuscation objective [41] pre-train objective, and its implementation in the C language. The authors would like to thank Srini Devadas and Slava Gerovitch, whose tireless efforts running the MIT PRIMES program enabled this research. Finally, the authors would like to thank Lindsey Lohwater and Rob Bauer for supporting Zifan Guo's participation in the MIT PRIMES research program.

This research was supported in part by the MIT PRIMES program under grant number 6946149. William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323. This research was supported in part by Los Alamos National Laboratories Grant 531711. Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator 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. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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 pretrained model for programming and natural languages, 2020.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec,

978-1-6654-9786-2/22/$31.00 ©2022 IEEE

and Guillaume Lample. Dobf: A deobfuscation pre-training objective for programming languages, 2021.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

978-1-6654-9786-2/22/$31.00 ©2022 IEEE