NEURAL INSTRUCTION COMBINER

Sandya Mannarswamy & Dibyendu Das
Intel India, Bangalore, india
{sandyasm, dibyendu.das0708}@gmail.com

ABSTRACT

Instruction combiner (IC) is a critical compiler optimization pass, which replaces a sequence of instructions with an equivalent and optimized instruction sequence at basic block level. There can be thousands of instruction-combining patterns which need to be frequently updated as new coding styles/idioms/applications and novel hardware evolve over time. This results in frequent updates to the IC optimization pass thereby incurring considerable human effort and high software maintenance costs. To mitigate these challenges associated with the traditional IC, we design and implement a Neural Instruction Combiner (NIC) and demonstrate its feasibility by integrating it into the standard LLVM compiler optimization pipeline. NIC leverages neural Seq2Seq model techniques for generating optimized encoded Intermediate Representation (IR) sequence from the unoptimized encoded IR sequence. To the best of our knowledge, ours is the first work demonstrating the feasibility of a neural instruction combiner built into a full-fledged compiler pipeline. Given the novelty of this task, we built a new dataset for training our NIC neural model. We show that NIC achieves exact match results percentage of 72% for optimized sequences as compared to traditional IC and Bleu precision score of 0.94, demonstrating its feasibility in a production compiler pipeline.

1 INTRODUCTION

Of late, considerable strides have been made in applying deep learning (DL) techniques to software engineering it-self, including source code assistance, automatic source code generation and in building software tools Le et al. (2020). The emergence of open-source community software development and large code repositories such as GitHub have accelerated interest in applying DL techniques to programming, compiler optimizations, code generation etc. Neural models have been developed for source code Alon et al. (2019) and intermediate code representations VenkataKeerthy et al. (2020). ML models have been used for cost prediction and heuristics selection in compiler optimizations Leather & Cummins (2020).

Instruction Combining pass is a basic compiler optimization pass present in all compilers. Instruction Combiner [IC] does local instruction level optimizations on basic blocks (BB) which are jump-free sequential lists of instructions. IC operates on the compiler's Intermediate Representation (IR) and replaces a sequence of one or more instructions with an optimized and semantically equivalent instruction sequence. ICs are typically developed with considerable human effort. There are thousands of patterns that are considered for replacement and these patterns continually need to be added/updated/removed as new coding styles/idioms/applications/hardware evolve over time. A typical IC pass often spans several thousand lines of code making it complex to maintain/debug/enhance. Empirical studies show that IC is the most frequently updated pass in the LLVM compiler Zhou et al. (2021).

Given the code complexity, software maintenance effort and its wide usage, IC is an ideal target for improvement with machine learnt models. However, there also exist considerable challenges in replacing a deterministic and human-written IC with a probabilistic machine learnt NIC. These challenges include representation of the input instruction sequence to the neural model, ensuring correctness of the probabilistic generated code by the neural model, integration of the neural model into a standard compiler optimization pipeline etc. This brings up the question of whether it is feasible to replace traditional IC by a neural model.

In this paper, we design and implement a Neural Instruction Combiner (NIC) and demonstrate its feasibility by integrating it into the standard LLVM compiler optimization pipeline which can generate executable machine code. NIC leverages neural Seq2Seq model techniques for generating optimized encoded IR sequence from the unoptimized encoded IR sequence at the basic block level, modelling it as monolingual machine translation task. We improve the standard attention mechanism with a compiler guided attention approach. NIC consists of three major components:

- **NIC Inputter** - This is a compiler module (not an ML model) which creates a distilled representation of the IR instruction sequence corresponding to each BB.
- **NIC Converter** - This is a machine learnt model which takes as input, an encoded representation of IR instruction sequence corresponding to each BB of a function and converts it to an equivalent optimized sequence. This model is trained offline and employed in inference mode in LLVM optimizer.
- **NIC Outputter** - This is a compiler module which takes as input the optimized sequence generated by NIC converter module and the original list of instruction corresponding to that BB and recreates the standard LLVM IR optimized instruction sequence for that BB. The output of NIC Outputter is then passed to the other downstream optimization passes. It also performs a set of IR verification checks and translation validity checking using Alive2 tool Lopes et al. (2021) to ensure semantic correctness of the NIC generated optimized IR sequence.

Given the novelty of this task, we create a new dataset for training our NIC neural model. We show that NIC can achieve exact match of 72% on optimized sequences as compared to traditional IC and a Bleu precision score of 0.94. To the best of our knowledge, ours is the first work demonstrating the feasibility of a neural instruction combiner built into a full-fledged compiler. We also outline the open challenges that need to be addressed.

Similar in spirit and complementary to our work is the work on building super optimizers from program binaries Bansal & Aiken (2006). They work by harvesting instruction sequences from binaries, enumerating their equivalent efficient target sequences by exhaustive search techniques, and creating an offline database of optimized instruction sequences. This work is limited to X86 instruction set. We also note that NIC can in fact leverage the binary instruction sequences harvested through super optimizers as training data by lifting up the binary instructions to LLVM IR using existing decompilation tools Cloutier.Next, we briefly discuss the background for our work.

2 BACKGROUND

The heart of any compiler is the optimizer which works on the compiler Intermediate Representation (IR) of the input program. A function in compiler's IR is split up into basic blocks (BB). Each BB is a sequence of IR instructions which do not modify the control flow and ends with a branch to another BB or a return instruction. IC attempts to replace a source sequence of one or more instructions by an equivalent and optimized sequence of instructions (IC can also replace single instructions with equivalent but optimized instructions). IC transformations typically include algebraic simplifications, instruction canonicalization, local constant propagation, constant folding etc.

In the IC pass the instruction sequence is scanned against multiple pattern-matching rules and once a sequence which matches the pattern is found, an equivalent and efficient transformed sequence of the identified pattern is applied to generate the new instruction sequence that replaces the original sequence. Appendix shows an example instruction sequence optimized by IC. Huge software code costs associated with IC and the fact that it is ubiquitous in all compilers opens the possibility of replacing the hand-coded rule driven pattern matching IC pass with a machine learnable IC pass.

3 DESCRIPTION

NIC consists of three major components namely **NIC Inputter**, **NIC Converter** and **NIC Outputter**. We briefly discuss each of these components next.

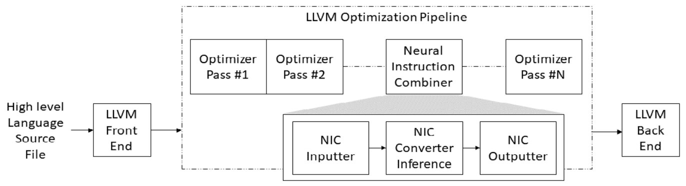

Figure 1: NIC Converter Inference Path

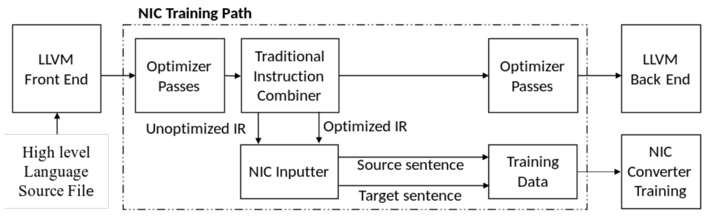

Figure 2: NIC Converter Training path

3.1 NIC Inputter

The input to the NIC Inputter is a regular full-fledged LLVM IR instruction stream corresponding to each BB of a function. NIC Inputter then creates a compressed encoded representation of the full LLVM IR instruction stream for that BB. An LLVM IR instruction contains information such as debug information, named variables, initializers, instruction level completers and metadata, which is not needed by IC itself. This leads us to create an encoded distilled representation of the full-fledged LLVM IR instruction sequence as input to the NIC Converter module. Using the distilled representation of the LLVM IR enables our NIC Converter to deal with a smaller vocabulary and makes it more efficient. Appendix-1 contains an example instruction sequence and its encoding. For each LLVM IR instruction, the distilled representation contains the target opcode, its type, each of the source operands and their types. In case a source operand is being produced by an instruction, the source operand is represented by its opcode. We concatenate the distilled encoded instructions for the BB and this becomes the input to NIC Converter.

3.2 NIC Converter

NIC Converter is a Sequence-to-Sequence (Seq2Seq) model which is modelled like a monolingual neural machine translator and is invoked in inference mode, during the LLVM optimizer pipeline by NIC pass as shown in Figure 1. This takes as test input, the encoded BB instruction sequence from NIC Inputter, and predicts an optimized encoded instruction sequence corresponding to it. The predicted encoded instruction sequence is then fed to the NIC Outputter.The source sentence is the encoded instruction sequence corresponding to a BB and the target sentence is the optimized sequence for that BB.

As is typical practice, our Seq2Seq models are based on the standard encoder-decoder framework with attention Sutskever et al. (2014). We consider two design choices for the encoder-decoder network, one based on recurrent neural network with a single head attention Bahdanau et al. (2014), and another based on standard transformer model with multi-head attention Vaswani et al. (2017). The input sentence is converted into a fixed length representation using the encoder from which the decoder emits the target sentence, one token at a time. Attention mechanism is employed to improve the ability of the Seq2Seq model to attend to the most relevant encoder outputs, when decoding each respective token.

3.3 Training of NIC Converter

For the current work, we train NIC Converter using data obtained from traditional IC similar to behavior cloning Bain & Sammut (1999). As shown in Figure 2, training data is generated in an offline phase by the compiler using the NIC Inputter module. Given a source file in high level language supported by the compiler, the compiler takes the input one function at a time and generates the BB level IR instruction sequences corresponding to each BB in the function. The compiler invokes the NIC Inputter on these unoptimized instruction sequences to obtain the encoded source sequences and then passes the original (unencoded) instruction sequences through the traditional (non-neural) IC phase. The optimized instruction sequences at the output of the traditional IC phase are then passed through NIC Inputter and encoded target sequences are obtained. The compiler maintains the BB level mapping between the unoptimized and optimized encoded sentence pairs and creates the list of sentence pair <unoptimized encoded instruction sequence, IC optimized encoded instruction sequence> corresponding to each BB. This is done at each function level. Given a set of source files in a high-level language, this process is repeated for all source files, and a list of sentence pairs are generated as training data for the NIC Converter by the compiler. We then train the Seq2Seq model to create the machine learnt NIC converter model, using the standard cross-entropy loss objective at each token level. We also consider a variant of standard Seq2Seq model, wherein we use compiler knowledge to guide the attention process, which we describe next.

3.4 Compiler Guided Attention Training

Attention enables the decoder to selectively consider relevant words of the source sentence when emitting each token in the target sentence. Standard attention is learnt with the indirect objective of improving the translation quality and is not learnt in a supervised manner with respect to word/phrase alignment between source/target sentence since direct word alignment information is not typically available for the training data in general. Hence it may not always correlate well with the alignment between source and target sentences. However, in case of NIC Converter, since the compiler has exact knowledge of which source instructions are responsible for generating the corresponding target instructions at each constituent token level, we leverage the compiler knowledge in improving the soft alignments learnt by the attention network.

During training data generation, a compiler guided attention matrix CA is created by the compiler for each BB. CA matrix terms are fixed attention scores provided by the compiler and are not learnt during training. For each source sentence, compiler has information of which source instruction tokens map to corresponding optimized target instruction tokens and uses this to set each element of the CA matrix. Each element CA[i, j] corresponds to the probability of whether the ith token in target sentence (of length T) is mapped to jth token in the source sentence and the total probability of one is distributed among the relevant mapped tokens while the non-relevant mapped tokens are set to zero. This is like hard attention with CA[i, j] being non-zero if target token 'i' is mapped to source instruction token 'j' by the compiler and else zero. We smooth this hard attention matrix with a small correction term delta, adding it to all zero terms and adjusting non-zero terms accordingly to maintain the row sum as 1. CA matrix has similar semantics and same dimension as the learnt attention matrix 'A' with standard attention. In case of single head attention, A is the single head cross attention weights, and in MHA, we use the decoder cross-attention weights of last layer's head 0 as the learnt attention matrix 'A'.

We use CA matrix to force the learnt attention weights 'A' to be closer to it during the training process by adding an additional loss term to the training objective. This is compiler attention mismatch loss term which is the divergence of the learnt attention weights in each training step from the compiler guided attention matrix CA. We model the Compiler Attention Mismatch (CAM) loss between CA and A matrices using the standard cross entropy loss function as below:

$$CAM\ Loss(CA, A) = -(1/T)\sum_i \sum_j CA[i][j] * log(A[i][j]) \tag{1}$$

$$Total\ Loss = Decoder\ Cross\ Entropy\ Loss\ +\ CAM\ Loss \tag{2}$$

Compiler guided attention is not used during inference, as we hypothesize that compiler guided attention will enable the model to learn the appropriate attention weights during training itself.

3.5 NIC Outputter

The NIC Outputter takes two inputs, (i) the predicted instruction sequence from the NIC Converter and (ii) the original unmodified full-fledged LLVM IR instruction stream corresponding to that BB. Given that NIC Converter predicts the most probable target instruction sequence, we also enforce specific checks in the NIC Outputter to ensure that the generated target sequence does not violate the compiler integrity checks. NIC Outputter performs a verification check first on the generated target sequence. It checks that for each instruction in the generated sequence, the number of operands corresponding to that opcode are correct, and that each operand has previously been defined in the generated instruction sequence, and that the last instruction in the sequence is a branch/return instruction. In case any of these conditions are violated, it discards the generated target instruction sequence and outputs the source instruction stream corresponding to that BB as is.

If the verification checks on the generated sequence are satisfied, then NIC Outputter takes the generated instruction sequence and applies it on the source instruction stream corresponding to that

BB to produce the transformed full LLVM IR instruction stream. If NIC Converter predicted output is the same as its input, (the NIC Converter has not found any suitable transformation for the given input sequence) NIC Outputter just reproduces the unmodified source IR sequence as is without any checks. NIC Outputter also invokes the LLVM function level verification. This checks that the CFG (Control Flow Graph) is valid, all instructions are associated with a BB and specific instruction level checking based on the instruction type (the types of operands of binary operator are of the same type, the Static Single Assignment Cytron et al. (1989) form is valid, shifts and logicals happen only on integral types etc). We then check for translation validity of the generated sequence by passing it through a well-known LLVM IR translation validity checker ALIVE2 Lopes et al. (2021). Translation validity checking ensures the semantic correctness and equivalence of the generated sequence to the original unoptimized sequence. ALIVE2 is a fully automatic bounded translation validation tool for LLVM that supports all of its forms of undefined behavior. ALIVE2 checks pairs of instruction sequences in LLVM IR for refinement using an SMT solver. Instruction sequences rejected by ALIVE2 as non-valid are rejected at the NIC Outputter.

4 Experimental Evaluation

4.1 Dataset Description

Given the novelty of our task, there are no readily available datasets which can be used to train the NIC Converter model. Hence, we build a new dataset for this task and plan to make it available publicly. The dataset consists of sentence pairs both from the same language of LLVM IR. The source sentence is the encoded distilled sequence of the LLVM IR instruction sequence of a BB at the input of traditional IC Pass. The target sentence is the encoded distilled sequence of that instruction sequence at the output of the traditional IC Pass. Training data is generated by invoking our modified version of LLVM compiler pipeline on the C/C++ application source files. For generating the training data, we used a collection of C/C++ source files from LLVM Application Test Suite and from Angha Bench Test Suite da Silva et al. (2021). The dataset size is $367K$ and contains 66% of unoptimized sequences (where source and target sentences are identical) and 34% optimized sequences (where source and target sentences are different).

4.2 Experimental Results

We evaluate several Seq2Seq models for NIC Converter as shown in Table 1. The two main network choices are RNN (Recurrent Neural Network) and the standard transformer models with Multi Head Attention(MHA) and greedy decoder (we did not see any significant change with beam decoder). We implemented various models in TensorFlow Python framework for evaluation. All parameters are initialized by uniform distribution over [-0:1; 0:1]. The mini-batch stochastic gradient descent algorithm is employed to train the model with batch size of 64 and number of training epochs being 10. Hyper-parameters were chosen based on experimentation with validation set. In addition, we use Adam optimizer with custom rate scheduler Vaswani et al. (2017). We do a train/validation/test data split of 90%, 5% and 5% respectively.

For NIC Converter inference performance, we report the metrics of Bleu precision Papineni et al. (2002) and Rouge scores Lin & Och (2004) in Table 1. However, our task being code optimization, requires generation of exact encoded representation (even a single wrong token will lead to incorrect code). Hence our task specific metric is comparison of Exact Match (EM) results for the entire instruction sequence for each BB between the predicted sequence and the ground truth. We show the EM results separately in last columns of Table 1 respectively for (a) EM(un-opt) where the ground truth is same as the input sequence (cases where IC performs no rewriting) and (b) EM (opt) where the ground truth is an optimized translation of the input sequence (optimized cases where IC performs rewrites).

Across all models, Bleu and Rouge scores are not significantly different, indicating their general translation capability (there were differences only beyond last 2 digits). As is expected, the EM percentage is much higher for the unoptimized sequences and is an indicator of model ability to reproduce the exact input sequence correctly. In case of optimized sequences, model's Exact Match is around 69%-73% indicating considerable room for further improvements. We find that transformer

Table 1: NIC Experimental Results

Model	Bleu	Rouge-1 r	Rouge-1 p	Rouge-2 r	Rouge-2 p	Rouge-l r	Rouge-l p	EM(unopt)	EM(opt)
Bidirectional LSTM: 3 layer encoder, unidirectional greedy decoder	0.93	0.98	0.90	0.96	0.91	0.97	0.93	0.93	0.68
Transformer with layers=4, dmodel=128, dff=512, heads=8	0.94	0.98	0.90	0.96	0.91	0.97	0.94	**0.94**	**0.72**
Transformer with layers=6, dmodel=512, dff=2048, heads=8	0.91	0.98	0.90	0.96	0.91	0.96	0.93	0.93	0.71
Transformer with layers=2, dmodel=512, dff=2048, heads=8	0.93	0.98	0.90	0.96	0.91	0.97	0.94	0.93	0.70
Transformer layers=4, dmodel=128, dff=512, heads=8 and no POS Emb	0.94	0.98	0.90	0.96	0.92	0.97	0.94	0.94	0.70
Transformer with layers=4, dmodel=512, dff=2048, heads=16	0.93	0.98	0.90	0.96	0.91	0.97	0.93	0.94	0.71
Bi-directional LSTM (3 layer encoder, 1 layer decoder) with guided compiler attention	0.93	0.98	0.89	0.96	0.91	0.96	0.94	0.93	0.70
Transformer with layers=4, dmodel=128, dff=512, heads=8 with compiler guided attention	0.94	0.98	0.90	0.96	0.92	0.97	0.93	0.94	0.72

model with 8 attention heads, 4 layers, and embedding dimension 128 has the best Exact Match results in our experiments.

We find in general that transformer models are better than RNN models by 2-3% of EM percentage. While compiler guided attention improved EM percentage for RNN model by 2%, it is still lower than that of MHA transformer models. Compiler guided attention had negligible impact on transformer models. We hypothesize that this may be due to MHA's ability to better capture word alignment information in multiple head attention weights compared to single head attention. We find that removing transformer positional encodings did not have any impact on model performance, similar to earlier works Clouatre et al. (2021). We did not see any improvement in performance by increasing the number of layers and feed forward layer dimensions in our transformer models.

Table 2: Exact Match Error Analysis

Type of Error	Occurrence
Incorrect Constant	42.3%
Opcode Mismatch	34.9%
Type Issue (Sext/Zext)	6.7%
Operand Mismatch	1.4%
Others	14.7%

4.3 Exact Match Error Analysis

We analyzed NIC Converter outputs to understand the model shortcomings. In case of optimized sequences, we find that NIC not only generates the optimized opcode, but also correctly fixes up the uses of the replaced opcode with the newly generated opcode, allowing us to hypothesize that the model has learnt the implicit use-def chains Cytron et al. (1989) in the encoded representation.

Table 2 shows the major reasons for exact match errors by NIC. One of the common mistakes the model exhibited was in generating correct values for synthesized constants for certain operations. A frequent LLVM IR instruction is the GetElementPtr (GEP) instruction used to get the address of a sub element of an aggregate data structure such as a 'struct' or 'array'. For nested aggregate data structures, a sequence of GEP operations is emitted with appropriate constants from base address. One of the IC optimization sequences is coalescing multiple GEP operations into a single GEP with modified constant indices as operands. We find that NIC had issues in synthesizing the GEP indices correctly. We reason that NIC is not able to learn the rules to compute the GEP offsets correctly based on individual GEP operations. It ends up reproducing the memorized frequent constant values as GEP indices generating erroneous sequence.

A similar problem was noticed in the case of 'Alloca' instruction sequence optimization where stack offsets constant values were generated incorrectly by NIC. These errors are caught during the verification of the generated code sequence in the NIC Outputter, dropping the suggested replacement from NIC from being applied. For frequently occurring/unique constants such as powers of two occurring in Shift instructions, the model outputs the correct constants both in optimized and unoptimized sequences. However, for arbitrary constants such as those occurring in GEP/Alloca operands, models end up making mistakes. The error analysis indicates that our current model does not handle synthesized constants in the instruction sequence well. This needs to be addressed in future work.

5 Related Work

Of late, there has been considerable interest in applying deep learning techniques to compilers in the areas of phase ordering Huang et al. (2019), selection of optimization heuristics Cummins et al. (2017) and as part of optimization itself in register allocation Das et al. (2019) and inlining Trofin et al. (2021). Machine learnt models have been used in optimization heuristics selection such as prediction of unroll factors Stephenson & Amarasinghe (2005), inlining decisions Simon et al. (2013), vectorization Mendis et al. (2019), Haj-Ali et al. (2020) etc.

Our work falls under the category of deploying ML models directly in compiler optimizations. Similar in spirit and complementary to our work, there has been work on building super optimizers by creating a database of possible optimized sequences from the binaries Bansal & Aiken (2006). These techniques work by harvesting instruction sequences from binaries, enumerating their equivalent efficient target sequences by exhaustive search techniques, and creating an offline database of optimized instruction sequences which can then be looked up for a given sequence of instructions. This line of work is limited to X86 instructions. There has been work on improving the brute force search for optimization sequences Schkufza et al. (2012) using random search and RL methods Bunel et al. (2016). NIC can leverage the optimized instruction sequences generated by super optimizers by lifting them to LLVM IR using decompilation techniques Cloutier (We plan to explore this in future work) and then using them for training the NIC Converter making them complementary to our work. Since our neural model operates on an encoded condensed IR representation, it is possible to port our NIC to any compiler if we can provide the NIC Inputter/Outputter modules which can convert from compiler's IR to NIC's encoded representation and vice versa. Unlike super optimizers which work on specific binary instruction sets, this allows wider portability.

6 Open Issues and Conclusion

In this paper, we explored the feasibility of replacing the traditional Instruction Combiner with a neural instruction combiner in a widely used production level compiler. We find that we were able to train a neural instruction combiner module and integrate it with LLVM compiler's optimizer pipeline.

There remain a number of challenges and open issues to be addressed in future work. 72% of optimization opportunities of traditional IC is currently realized by NIC. Hence this gap in performance between NIC and traditional IC needs to be addressed by adding more relevant training samples.

In addition to training data generated from traditional IC, we also plan to expand NIC training data with IR instruction sequences that can be leveraged from existing super-optimizer databases. NIC does behavioral cloning of traditional IC currently. Another possibility is to use NIC to explore the optimized translation subspace by generating multiple translations and ranking the sequences based on their optimization potential and choosing the highest ranked one as the output translation. In this way, NIC can be integrated into super optimizers to effectively explore the optimized output sequence subspace.

Validity checks of generated sequence currently include compiler driven IR Validity checks and translation validity checking using ALIVE2 external tool Lopes et al. (2021). Leveraging scalable automatic post-editing Chollampatt et al. (2020) and program repair techniques Gupta et al. (2020) can help improve validity checks in NIC. These need to be addressed further for a robust NIC deployment in production compilers.

REFERENCES

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

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

Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15, Intelligent Agents [St. Catherine's College, Oxford, July 1995]*, pp. 103–129, GBR, 1999. Oxford University. ISBN 0198538677.

Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGPLAN Not.*, 40(5):394–403, October 2006. ISSN 0163-5980.

Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. *CoRR*, abs/1611.01787, 2016.

Shamil Chollampatt, Raymond Hendy Susanto, Liling Tan, and Ewa Szymanska. Can automatic post-editing improve nmt? *arXiv preprint arXiv:2009.14395*, 2020.

Louis Clouatre, Prasanna Parthasarathi, Amal Zouaq, and Sarath Chandar. Demystifying neural language models' insensitivity to word-order. *CoRR*, abs/2107.13955, 2021.

Felix Cloutier. Fcd an optimizing decompiler for x86. URL `https://github.com/zneak/fcd`.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *Proceedings - 26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017*, Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, pp. 219–232. IEEE, October 2017.

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pp. 25–35, New York, NY, USA, 1989. Association for Computing Machinery.

A. da Silva, B. Kind, J. de Souza Magalhaes, J. Rocha, B. Ferreira Guimaraes, and F. Quinao 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)*, pp. 378–390. IEEE Computer Society, 2021.

Dibyendu Das, Shahid Asghar Ahmad, and Kumar Venkataramanan. Deep learning-based hybrid graph-coloring algorithm for register allocation. *CoRR*, abs/1912.03700, 2019.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute and debug: Learning to repair for neural program synthesis. *arXiv preprint arXiv:2007.08095*, 2020.

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. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, pp. 242–255, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479.

Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 308–308, 2019.

Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Comput. Surv.*, 53(3), June 2020.

Hugh Leather and Chris Cummins. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–8, 2020.

Chin-Yew Lin and Franz Josef Och. Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pp. 605–612, Barcelona, Spain, July 2004.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pp. 65–79, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912.

Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. *Compiler Auto-Vectorization with Imitation Learning*. Curran Associates Inc., Red Hook, NY, USA, 2019.

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 of the Association for Computational Linguistics*, pp. 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. *CoRR*, abs/1211.0557, 2012.

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, pp. 1–12, USA, 2013. IEEE Computer Society.

M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*, pp. 123–134, 2005.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: a machine learning guided compiler optimizations framework. *CoRR*, abs/2101.04808, 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. Ir2vec - llvm ir based scalable program embeddings. *ACM Trans. Archit. Code Optim.*, 17(4), December 2020. ISSN 1544-3566.

Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. An empirical study of optimization bugs in gcc and llvm. *Journal of Systems and Software*, 174:110884, 2021. ISSN 0164-1212.