Instruction Combiner a critical pass in all modern compilers

Thousands of instruction-combining patterns

Patterns need to be frequently updated over time as software coding patterns/idioms/applications evolve

IC is the most frequently updated component in the LLVM compiler [Zhou et al. 2020].

Considerable human effort, high software maintenance costs

Is it possible to replace traditional IC with a machine learnt model?
Learning to Combine Instructions

• Can we replace the hand-coded rule driven pattern matching IC pass with a machine learnable IC pass?

• Modelled as monolingual machine translation task

• Neural Machine Translation (NMT) translates from source to target language

• Both source and target languages are LLVM Instruction IR

• We leverage neural Seq2Seq models for this task
  • State of art models using LSTMs and Transformers
Design choices for NIC

- What should be the input sentence encoding for Seq2Seq model?
- How can we find/build a dataset for this task?
- How do we integrate a machine learnt IC module into the overall pipeline?
- How do we validate the IR generated from NIC?
• NIC has three major components

• **NIC inputter**: (non-ML) creates an encoded representation from LLVM IR instruction corresponding to a basic block

• **NIC Converter**: (Seq2Seq Neural network model) takes the output from NIC Inputter and generates an equivalent optimized encoded instruction sequence

• **NIC Outputter**: (non-ML) converts the NIC Converter output back to full-fledged LLVM IR instruction sequence of a basic block. It also performs a set of IR verification checks and translation validity checking
Overview of Seq2Seq Models

**Encoder**
- Produces an encoding of the source sentence.
- Provides initial hidden state for **Decoder**.

**Decoder**
- Is a Language Model that generates target sentence, **conditioned on** encoding.

Note: This diagram shows test time behavior: decoder output is fed in as next step’s input.
• Vanilla Seq2Seq models have the information bottleneck problem due to single encoder output vector

• **Attention** provides a solution to the bottleneck problem

• **Core idea**: on each step of the decoder, use *direct connection to the encoder* to *focus on a particular part* of the source sequence

• Attention significantly **improves NMT performance**
  
  • It’s very useful to allow decoder to focus on certain parts of the source
Seq2Seq models typically contain an encoder, decoder and attention mechanism.

- Encoder creates a distilled representation of input.
- Decoder generates the output based on the encoder outputs and each previously generated output symbol.
- Attention weights selectively weigh the encoder outputs.
- Each encoder/decoder block can be a RNN (LSTM) or a transformer block (Multihead Attention).
NIC Building Blocks

• **NIC inputter** is the input (non-ML) module for NIC
  • Creates a distilled representation of the IR instruction sequence for each BB

• **NIC Converter** (ML Module)
  • Model trained offline and employed in inference mode in optimizer pipeline
  • Two variants: RNN based and Transformer based
  • NIC Converter uses two attention mechanisms
    • Standard attention mechanism of Seq2Seq models
    • A novel Compiler guided attention mechanism

• **NIC outputter** is the output module (non-ML) for NIC
  • Takes the NIC converter output along with source BB instruction list
  • Validates the instruction stream and emits the optimized IR instruction list
NIC Converter – RNN Based
NIC Converter – Transformer Based
LLVM Optimization Pipeline

High level Language Source File

LLVM Front End

Optimizer Pass #1

Optimizer Pass #2

Traditional Instruction Combiner

Optimizer Pass #N

LLVM Back End

Un-optimized IR

Optimized IR

NIC Inputter

Source Sentence

Target Sentence

Training Data

NIC Converter Training
Leverage the compiler knowledge in improving the soft attention alignments.

During training data generation, a compiler guided attention matrix CA is created.

CA matrix terms are fixed attention scores provided by the compiler and are not learnt during training.

Each element CA[i, j] corresponds to the probability of whether the $i^{th}$ token in target sentence maps to $j^{th}$ token in source sentence.

Force the learnt attention weights to be closer to CA during the training process.

- by adding an additional loss term to the training objective
• Created 300K samples dataset from LLVM application test suite & AnghaBench

• Trained the NIC seq2seq models using mini-batch gradient descent

• standard cross-entropy loss and Adam optimizer

• The trained NIC converter was then deployed in inference mode in the optimizer pipeline

• Evaluated with test data set

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>LSTM 3-layer bidirectional stacked encoder with 3-layer unidirectional greedy decoder.</td>
</tr>
<tr>
<td>B</td>
<td>Transformer: num_layers = 4, d_model = 128, dff = 512, num_heads = 8, dropout_rate = 0.1</td>
</tr>
<tr>
<td>C</td>
<td>Transformer: num_layers = 6, d_model = 512, dff = 2048, num_heads = 8, dropout_rate = 0.1</td>
</tr>
<tr>
<td>D</td>
<td>Same as B, with num layers = 2</td>
</tr>
<tr>
<td>E</td>
<td>Same as B, with No POS Embedding</td>
</tr>
<tr>
<td>F</td>
<td>Same as B, with 16 heads</td>
</tr>
<tr>
<td>G</td>
<td>Model A with compiler guided attention</td>
</tr>
<tr>
<td>H</td>
<td>Model B with compiler guided attention</td>
</tr>
</tbody>
</table>
• Standard Machine Translation metrics are Bleu s & Rouge Scores
  • BLEU evaluates the quality of translation, a number between 0 to 1
    • 1 -> machine translation and human translation were identical.
    • Bleu precision evaluated at multiple n-gram level with average across all n-gram levels being reported as a single final score.
  • Rouge-n score represents the n-gram overlap between the machine generated and ground truth reference translations
• Task specific metric is Exact Match (EM) comparison results
  • for each BB between the predicted sequence and the ground truth
  • Reported separately for optimized and unoptimized sequences
<table>
<thead>
<tr>
<th>Metric</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bleu precision</td>
<td>0.93</td>
<td>0.94</td>
<td>0.91</td>
<td>0.93</td>
<td>0.94</td>
<td>0.93</td>
<td>0.93</td>
<td>0.94</td>
</tr>
<tr>
<td>Rouge-1 r score</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>Rouge-1 p score</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.89</td>
<td>0.90</td>
</tr>
<tr>
<td>Rouge-2 r score</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td>Rouge-2 p score</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.92</td>
<td>0.91</td>
<td>0.91</td>
<td>0.92</td>
</tr>
<tr>
<td>Rouge-l r score</td>
<td>0.97</td>
<td>0.97</td>
<td>0.96</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>Rouge-l p score</td>
<td>0.93</td>
<td>0.94</td>
<td>0.93</td>
<td>0.94</td>
<td>0.94</td>
<td>0.93</td>
<td>0.94</td>
<td>0.93</td>
</tr>
<tr>
<td>Exact Match (un-opt)</td>
<td>0.93</td>
<td>0.94</td>
<td>0.93</td>
<td>0.93</td>
<td>0.94</td>
<td>0.94</td>
<td>0.93</td>
<td>0.94</td>
</tr>
<tr>
<td>Exact Match (opt)</td>
<td>0.68</td>
<td>0.72</td>
<td>0.71</td>
<td>0.70</td>
<td>0.70</td>
<td>0.71</td>
<td>0.70</td>
<td>0.72</td>
</tr>
</tbody>
</table>
• NIC correctly fixes up the uses of the replaced opcode with the newly generated opcode

• For frequent/unique constants (Shift instructions), the model outputs the correct constants

• Mistakes in generating correct values for synthesized constants
  • such as GEP and Alloca operands
  • ends up reproducing the memorized frequent constant values

---

<table>
<thead>
<tr>
<th>Type of error</th>
<th>Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect Constant</td>
<td>42.3%</td>
</tr>
<tr>
<td>Opcode Mismatch</td>
<td>34.9%</td>
</tr>
<tr>
<td>Type issue (Sign/Zero extension)</td>
<td>6.7%</td>
</tr>
<tr>
<td>Operand swap (canonicalization)</td>
<td>1.4%</td>
</tr>
<tr>
<td>Others</td>
<td>14.7%</td>
</tr>
</tbody>
</table>
Recent work in applying deep learning techniques to compilers
- Optimization phase ordering
- Selection of optimization heuristics
- Compiler cost models

Building super optimizers for binaries [Bansal 2006]
- Creating a database of possible optimized sequences from the binaries
- Limited to X86 binaries
- Incur high overheads due to huge candidate search space
- Improving Super Optimizers [Schkufza 2013, Bunel 2017]
Only 72% of optimization opportunities are realized by NIC

Correctness checks for NIC generated code sequences
  - IR and CFG Validation Checks
  - Use of ALIVE2 for translation validity checking
  - Automatic NMT post editing techniques/Program repair techniques in future?

Expanding the dataset for training NIC
  - Currently learning from the traditional IC (behavioral cloning)
  - Leverage super optimizer identified instances in future?