POSET-RL: Phase ordering for Optimizing Size and Execution Time using Reinforcement Learning

Shalini Jain, Yashas Andaluri, S. VenkataKeerthy, Ramakrishna Upadrasta

Scalable Compilers for Heterogeneous Architectures Lab
Indian Institute of Technology Hyderabad

https://compilers.cse.iith.ac.in/

LLVM-CGO 2022
Apr 03, 2022
Phase Ordering of Compiler Optimizations

- Find optimal sequence of optimization passes to improve code performance

Why is it Important?
- One optimization sequence does not guarantee improvement for all programs
- Different permutations of an optimization sequence may yield different performances.
Trade-off: Code Size vs. Execution Time

O0: No optimizations

O1, O2, O3: Improve performance by reducing execution time
May increase code size

Os: Execution time nearly equal to O2
Reduces code size

Oz: Designed to provide more code size reduction
May increase execution time

Trade-off between O3 and Oz in terms of execution time and code size
Phase Ordering for Code Size and Execution Time

Problem with single objective

● Optimizing only for code size may adversely affect execution time
  ○ can ignore passes: unrolling, inlining

● Optimizing only for execution time may adversely affect code size
  ○ can aggressively unroll or inline

Dual objective

● Co-optimize code size and execution time
O3 vs. Oz: Comparison of runtime and code size
O3 vs. Oz: Comparison of runtime and code size
POSET-RL - Overview

- Reinforcement Learning model
  - Predicts the optimal sequences of passes for a given program
  - Optimizes program for both size and execution time

- Builds from the embeddings given by IR2Vec framework
  - Represents program as a higher dimensional vectors
  - Encodes program features, flow information and semantics

POSET-RL - Overview

- Predictions: sub-sequences of optimization passes
  - Derive sub-sequences manually from Oz
  - Generate sub-sequences from *Oz Dependence Graph (ODG)*
    - ODG: Graph formed from -Oz pass sequence

- Architecture neutral approach
  - Results on X86 and AArch architectures
Reinforcement Learning

- Basic blocks of Reinforcement Learning models
  - Environment
  - State
  - Agent
  - Action
  - Reward
Why is Phase ordering an RL problem?

- For Oz
  - No. of transformation passes = 90
  - No. of unique transformation passes = 54
  - $54^{90} \approx 10^{156}$ combinations are possible
Proposed Workflow/Methodology
Environment and State

- Agent interacts with environment and produces new state
- IR2Vec Embeddings acts as a state
- Two different approaches for action space
  - Manual Selection of Subsequences
  - Subsequence generation by Oz Dependence Graph (ODG)
Sub-sequences Generated by Manual Grouping

- Sub-sequences created from LLVM’s Oz sequence
  - Manually created 15 sub-sequences
- Group the passes according to their functionality
  - Loop passes, global optimizations separated into their own sub-sequence
- Not easy to tune sub-sequences manually
  - Requires knowledge of each pass
<table>
<thead>
<tr>
<th>S. No.</th>
<th>Manual Sub-sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-ee-instrument -simplifycfg -sroa -early-cse -lower-expect -forceattrs -inferattrs -mem2reg</td>
</tr>
<tr>
<td>2</td>
<td>-ipsccp -called-value-propagation -attributor -globalopt</td>
</tr>
<tr>
<td>3</td>
<td>-deadargelim -instcombine -simplifycfg</td>
</tr>
<tr>
<td>4</td>
<td>-prune-eh -inline -functionattrs -barrier</td>
</tr>
<tr>
<td>5</td>
<td>-sroa -early-cse-memssa -speculative-execution -jump-threading -correlated-propagation</td>
</tr>
<tr>
<td>6</td>
<td>-simplifycfg -instcombine -tailcalleeelim -simplifycfg -reassociate</td>
</tr>
<tr>
<td>7</td>
<td>-loop-simplify -lcssa -loop-rotate -licm -loop-unswitch -simplifycfg -instcombine</td>
</tr>
<tr>
<td>8</td>
<td>-loop-simplify -lcssa -indvars -loop-idiom -loop-deletion -loop-unroll</td>
</tr>
<tr>
<td>9</td>
<td>-mldst-motion -gvn -memcpyopt -scce -instcombine -jump-threading -correlated-propagation -dse</td>
</tr>
<tr>
<td>10</td>
<td>-loop-simplify -lcssa -licm -adce -simplifycfg -instcombine</td>
</tr>
<tr>
<td>11</td>
<td>-barrier -elim-avail-extern -rpo-functionattrs -globalopt -globaldce -float2int -lower-constant-intrinsics</td>
</tr>
<tr>
<td>12</td>
<td>-loop-simplify -lcssa -loop-rotate -loop-distribute -loop-vectorize</td>
</tr>
<tr>
<td>13</td>
<td>-loop-simplify -loop-load-elim -instcombine -simplifycfg -instcombine</td>
</tr>
<tr>
<td>14</td>
<td>-loop-simplify -lcssa -loop-unroll -instcombine -loop-simplify -lcssa -licm -alignment-from-assumptions</td>
</tr>
<tr>
<td>15</td>
<td>-strip-dead-prototypes -globaldce -constmerge -loop-simplify -lcssa -loop-sink -instsimplify -div-rem-pairs -simplifycfg</td>
</tr>
</tbody>
</table>
ODG: Oz Dependence Graph

- Constructed from Oz pass sequence
  - Each individual optimization pass => Node of the graph
  - If pass A precedes pass B in Oz sequence, then Add edge: A -> B
- Critical node: node with degree >= k (k = 8)
- Subsequence: walk that starts and ends at a critical node
Oz Dependence Graph (ODG)
Sub-sequences generated by Oz Dependence Graph (ODG)
Sub-sequences generated by Oz Dependence Graph (ODG)
Significance of ODG sub-sequences

- Designing sub-sequences manually may not include all possible orders
- Uncovers new sub-sequences not present in Oz
- Preserves ordering of passes in Oz
- In total, 34 sub-sequences are generated with 3 critical nodes
**Reward Computation**

$$R = \alpha \cdot R_{\text{BinSize}} + \beta \cdot R_{\text{Throughput}}$$

- **Reward for Binary Size**: $$R_{\text{BinSize}} = \frac{\text{BinSize}_{\text{last}} - \text{BinSize}_{\text{curr}}}{\text{BinSize}_{\text{base}}}$$
- **Reward for Execution Time**: $$R_{\text{Throughput}} = \frac{\text{Throughput}_{\text{curr}} - \text{Throughput}_{\text{last}}}{\text{Throughput}_{\text{base}}}$$

- $$\alpha = 10$$
- $$\beta = 5$$

**Static measure of runtime**

Computed by LLVM-MCA
### Training

- **Intel Xeon E5-2690 and Intel Gold 5122**

**Parameters:**
- Learning rate: $10^{-4}$
- #time steps per iteration: 1005
- 16 hours to train

**Dataset:**
- 130 files from single source benchmarks from LLVM-Test-Suite

**Double Deep Q-Network (DDQN) Algorithm**

### Inference

- **X86 architecture**
  - Intel Xeon E5-2697

- **AArch architecture**
  - Cross compiling LLVM to target Cortex-A72 processor

**Results:**
- SPEC-CPU-2017
- SPEC-CPU-2006
- MiBench
Results: Percentage Code-Size Reduction

Percentage of min, avg and max size reduction with manual and ODG sequences wrt Oz

x86

AArch64

% Size Reduction

SPEC2017  SPEC2006  MiBench

SPEC2017  SPEC2006  MiBench
Results: Percentage Execution-Time Improvement

Percentage of improvement in execution time with manual and ODG sequences wrt Oz for X86
Results: Binary Size for SPEC
Results: Execution Time for SPEC
Summary

● A RL based framework to solve Phase Ordering problem
  ○ Improves both code size and execution time

● Model action space by two approaches
  ○ Manual sub-sequences
  ○ ODG sub-sequences

● Rewards: static measure of codesize and runtime

● Results on X86 and AArch

● ODG can be extended to O3 (execution time)

To appear in ISPASS 2022
https://compilers.cse.iith.ac.in/projects/posetrl/
Thank You