AUTOPHASE: JUGGLING HLS PHASE ORDERINGS IN RANDOM FORESTS WITH DEEP REINFORCEMENT LEARNING

Ameer Haj-Ali Qijing Huang William Moses John Xiang Krste Asanovic John Wawrzynek Ion Stoica

Presenter: Chengzhe Li

Outline

Introduction

Related Works

Approach

Experiments

Conclusion

Introduction

Problem

- HLS(High-Level Synthesis) also needs compiler optimization!
- Phase ordering problem

Difficulty

- NP-Hard problem(O(n!) or O(n^n))
- Phases are not commutative
- In this work the search space extends to more than 2^247 phase orderings

Good example

loop invariant code motion (LICM)

Optimization first, then inlining

```
__attribute__((const))
double mag(int n, const double *A) {
    double sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    return sqrt(sum);
}
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);
    }
}
```

Figure 1: A simple program to normalize a vector.

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double precompute = mag(n, in);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double precompute, sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    precompute = sqrt(sum);
    for(int i=0; i<n; i++) {
        out[i] = in[i] / precompute;
    }
}
```

Figure 2: Progressively applying LICM (left) then inlining (right) to the code in Figure 1.

Bad example

Inlining first, then LICM

```
__attribute__((const))
double mag(int n, const double *A) {
    double sum = 0;
    for(int i=0; i<n; i++){
        sum += A[i] * A[i];
    }
    return sqrt(sum);
}
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);
    }
}
```

Figure 1: A simple program to normalize a vector.

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    for(int i=0; i<n; i++) {
        double sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

Problem!

```
void norm(int n, double *restrict out,
          const double *restrict in) {
    double sum;
    for(int i=0; i<n; i++) {
        sum = 0;
        for(int j=0; j<n; j++){
            sum += A[j] * A[j];
        }
        out[i] = in[i] / sqrt(sum);
    }
}
```

Figure 3: Progressively applying inlining (left) then LICM (right) to the code in Figure 1.

This paper

- This problem is suitable for RL algorithm
- deep reinforcement learning (RL) (Sutton & Barto, 1998; Haj-Ali et al., 2019b) to address the phase ordering problem
- Each action can change the state of the environment and generate a ”reward”, in this case, improvement.
- Goal: a policy between states and actions that maximize the reward
- Deep RL algorithm: a DNN to approximate the policy
- An importance analysis on the features using random forests
- Build off LLVM compiler
- AutoPhase: a framework that integrates the current HLS compiler infrastructure with the deep RL algorithms.

Approaches

1. directly use salient features from the program
2. derive the features from the sequence of optimizations we applied while ignoring the program’s features
3. combines the first two approaches

Backgrounds

- Compiler Phase ordering (adjustable levels)
  - Heuristics with machine learning
  - Modified Greedy algorithm
  - Machine learning to decide passes to apply
  - autotunes the program
  - Supervised learning
- Reinforcement Learning algorithm

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim \pi(\tau)} \left[\sum_t r(s_t, a_t) \right] =$$

$$\arg\max_{\pi} \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim \pi(s_t, a_t)} \left[r(s_t, a_t) \right].$$

Background (Cont.)

- PG(Policy Gradient)

  On-Policy method, use the decision made by current policy directly to update new policy

- PPG(Proximal Policy Gradient)

  enables multiple epochs of minibatch updates to improve the sample complexity

  Ensure the deviation from previous to be small while maximizing the reward function

- A3C(Asynchronous Advantage Actor-critic)

  Used to determines whether an action is good or not and how to adjust

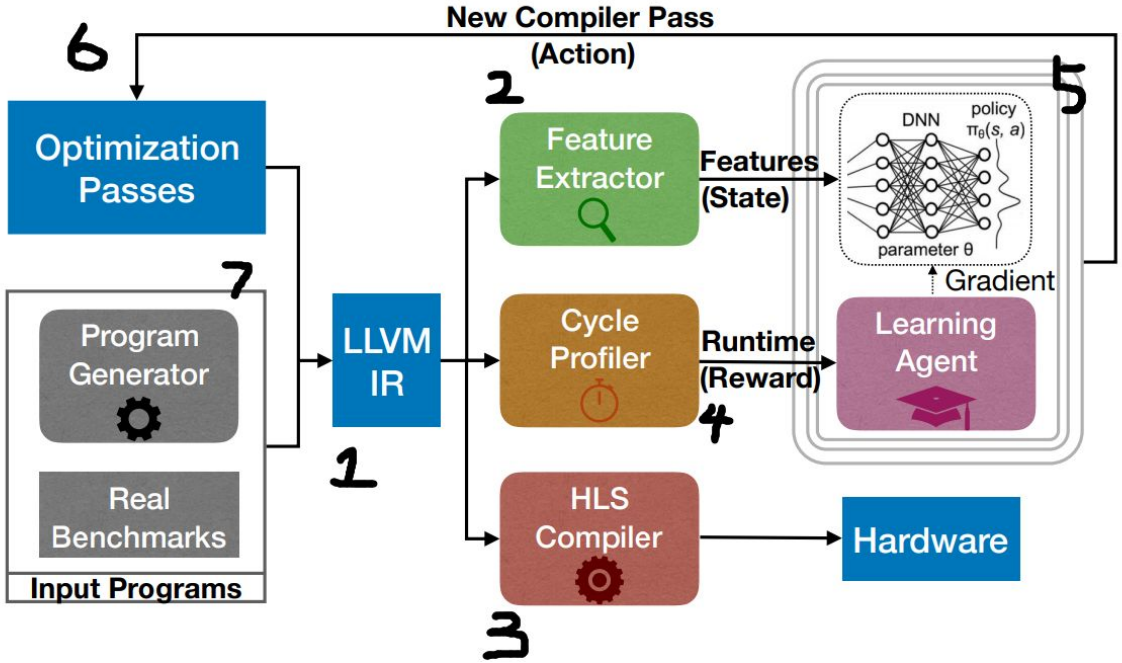
Evolutionary Algorithm

- Used to displace Gradient-based algorithms
- Genetic Algorithms (GA)
- Evolution Strategies (ES)

Approach 1. Implementation

- LLVM Compiler:Program to IR (simple compiler)
- LegUp: HLS framework(tool), LLVM IR-> hardware RTL design (instructions)
- Clock-cycle Profiler (approximating hardware simulation)
- IR Feature Extractor(Wang et al. (Wang & OBoyle, 2018))
- Two random forests
- Random Program Generator(Generating training set)
- Possible extensibility

Workflow



1. The input program is compiled into LLVM IR using the Clang/LLVM.
2. The IR Feature Extractor is run to extract salient program features.
3. LegUp compiles the LLVM IR into hardware RTL.
4. The Clock-cycle Profiler estimates a clock-cycle count for the generated circuit.
5. The RL agent takes the program features or the histogram of previously applied passes and the improvement in clock-cycle count as input data to train on.
6. The RL agent predicts the next best optimization passes to apply.
7. New LLVM IR is generated after the new optimization sequence is applied.
8. The machine learning algorithm iterates through steps (2)–(7) until convergence.

Table 1: LLVM Transform Passes.

0	1	2	3	4	5	6	7	8	9	10	
-correlated-propagation	-scalarrepl	-lowerinvoke	-strip	-strip-nondebug	-sccp	-globalopt	-gvn	-jump-threading	-globaldce	-loop-unswitch	
11	12	13	14	15	16	17	18	19	20	21	
-scalarrepl-ssa	-loop-reduce	-break-crit-edges	-loop-deletion	-reassociate	-lcssa	-codegenprepare	-memcpyopt	-functionattrs	-loop-idiom	-lowerswitch	
22	23	24	25	26	27	28	29	30	31	32	33
-constmerge	-loop-rotate	-partial-inliner	-inline	-early-cse	-indvars	-adce	-loop-simplify	-instcombine	-simplifycfg	-dse	-loop-unroll
34	35	36	37	38	39	40	41	42	43	44	45
-lower-expect	-tailcallelim	-licm	-sink	-mem2reg	-prune-eh	-functionattrs	-ipsccp	-deadargelim	-sroa	-loweratomic	-terminate

Table 2: Program Features.

0	Number of BB where total args for phi nodes >5	28	Number of And insts
1	Number of BB where total args for phi nodes is [1,5]	29	Number of BB's with instructions between [15,500]
2	Number of BB's with 1 predecessor	30	Number of BB's with less than 15 instructions
3	Number of BB's with 1 predecessor and 1 successor	31	Number of BitCast insts
4	Number of BB's with 1 predecessor and 2 successors	32	Number of Br insts
5	Number of BB's with 1 successor	33	Number of Call insts
6	Number of BB's with 2 predecessors	34	Number of GetElementPtr insts
7	Number of BB's with 2 predecessors and 1 successor	35	Number of ICmp insts
8	Number of BB's with 2 predecessors and successors	36	Number of LShr insts
9	Number of BB's with 2 successors	37	Number of Load insts
10	Number of BB's with >2 predecessors	38	Number of Mul insts
11	Number of BB's with Phi node # in range (0,3]	39	Number of Or insts
12	Number of BB's with more than 3 Phi nodes	40	Number of PHI insts
13	Number of BB's with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Number of occurrences of 64-bit integer constants	48	Number of Xor insts
21	Number of occurrences of constant 0	49	Number of ZExt insts
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations

Approach 2. Problem Formulation

Action Space: [0,K] (total number of passes)

Observation Space: Features and Action History

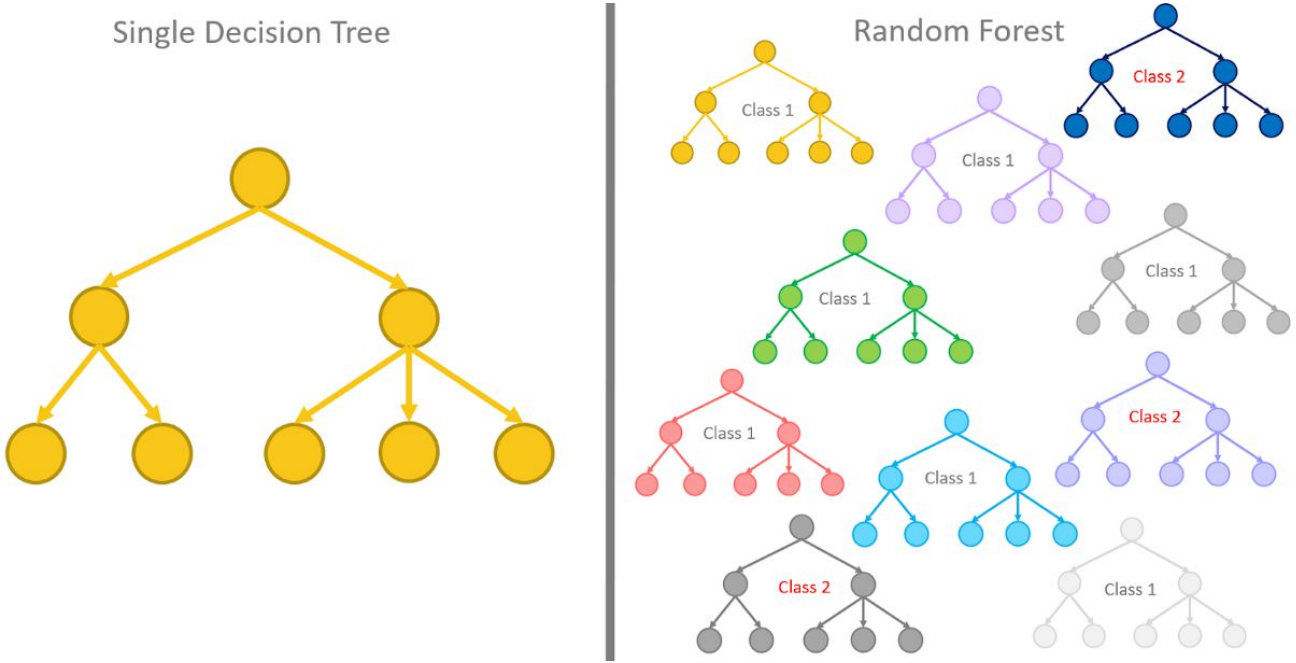
Reward: R = c(cycle count) curr - c previous

Multiple passes pre action([-1,0,1], change to new pass or not)

Normalization: Taking the logarithm and normalizing to a parameter

Details

- Use Random Forests to learn the importance of features



Example of feature importance

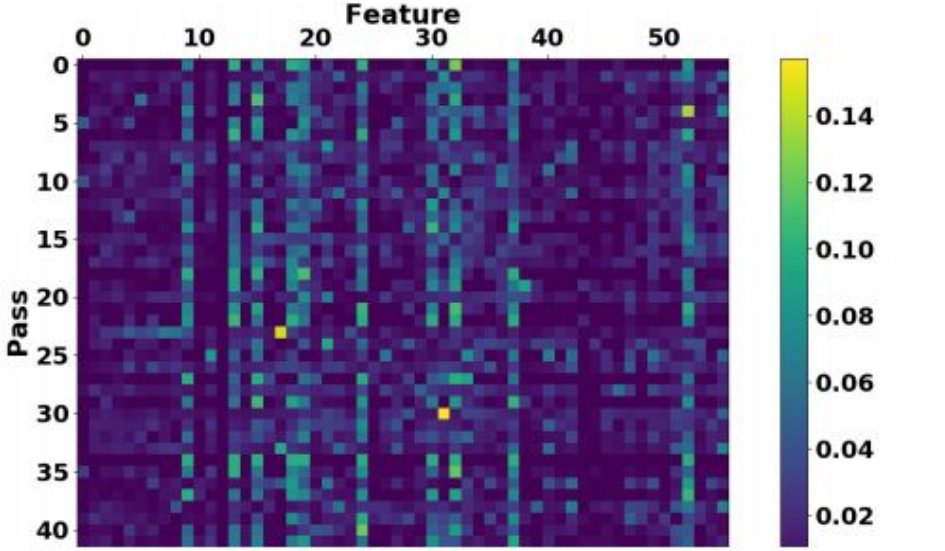


Figure 5: Heat map illustrating the importance of feature and pass indices.

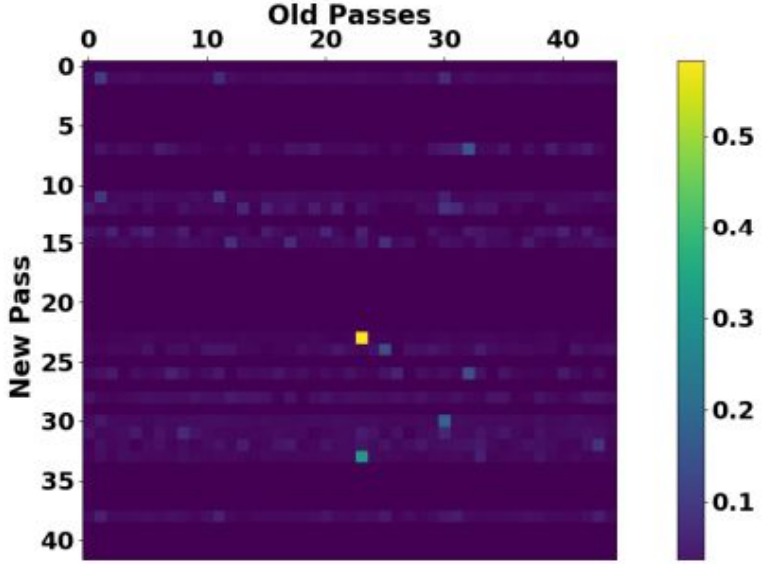


Figure 6: Heat map illustrating the importance of indices of previously applied passes and the new pass to apply.

Experiments: Setups

- RLlib, a open-source library for Reinforcement Learning built on top of Ray
- Ray, a high performance distributed execution framework
- Environment: 4 cores Intel i7-4765T CPU with a Tesla K20c GPU
- frequency constraint in HLS: 200MHz
- Metrics: clock cycles reported in HLS profiler
- Previous research shows the linear relation between clock cycle and hardware performance
- Run on 9 real HLS benchmarks and compare performances
- Compared: random search, greedy, OpenTuner, Genetic Algorithm ...

Table 3: The observation and action spaces used in the different deep RL algorithms.

	RL-PPO1	RL-PPO2	RL-PPO3	RL-A3C	RL-ES
Deep RL Algorithm	PPO	PPO	PPO	A3C	ES
Observation Space	Program Features	Action History	Action History + Program Features	Program Features	Program Features
Action Space	Single-Action	Single-Action	Multiple-Action	Single-Action	Single-Action

Performance

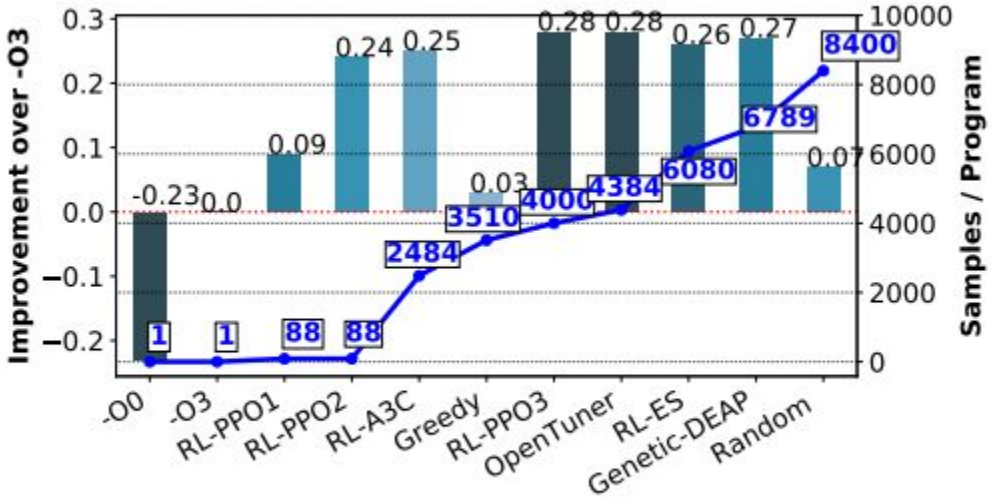


Figure 7: Circuit Speedup and Sample Size Comparison.

Generalization test using 100 Randomly generated programs

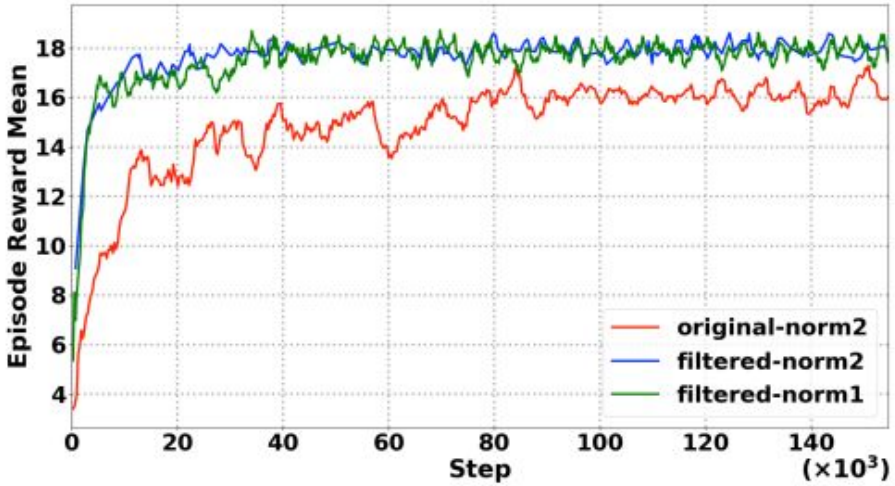


Figure 8: Episode reward mean as a function of step for the original approach where we use all the program features and passes and for the filtered approach where we filter the passes and features (with different normalization techniques). Higher values indicate faster circuit speed.

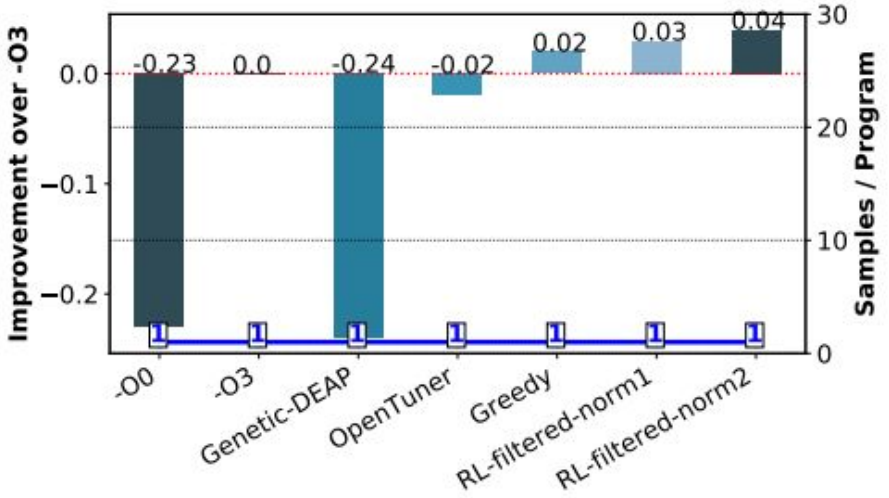


Figure 9: Circuit Speedup and Sample Size Comparison for deep RL Generalization.

Conclusion

Pros

1. The idea is worth of further study
2. Formulated the compiler optimization into machine learning problem

Cons

1. Decoupled features and passes in an lossy way
2. Lack of both training and testing data
3. Improvement is not huge

Question?

Thank you!