Learning to Represent Programs with Graphs

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi

Presenter: Hee Hwang (Peter) 11/04/2020

Agenda (~15 minutes)

1. Problem / Related Work
2. Tasks
3. Model
4. Data / Evaluation
5. Limitation / Discussion

Agenda

1. Problem / Related Work

RNN on formal language(source code)

```
/*
 * Increment the size file of the new incorrect UI_FILTER group information
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
```

Cribbed from The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Basic Idea

Variables: ♠, ♣, ♥, ♦

❓= foobar(7)

Answer:

♠: 60%, ♣: 22%, ♥: 10%, ♦: 8%

Practical Example

```
var clazz=classTypes["Root"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

var first=classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull( clazz );

Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

Figure 1: A snippet of a detected bug in RavenDB an open-source C# project. The code has been slightly simplified. Our model detects correctly that the variable used in the highlighted (yellow) slot is incorrect. Instead, `first` should have been placed at the slot. We reported this problem which was fixed in PR 4138.

ML for source code artifacts

- Model the code as a sequence of tokens
- Model the syntax tree structure of code
- Predict variable names using all their usage
- CRF for variables, AST elements, and types

Problems of previous approach

- Does not exploit formal structure of source code
- No Data Flow Analysis
- Specification required
  - Input-output Examples
  - Test Suites

Agenda

1. Problem / Related Work
2. **Tasks**

VarNaming & VarMisuse

- **VarNaming:**
  - Predict "correct" variable name from the given some source code
  - Analogous to filling the blank

- **VarMisuse:**
  - Detect variable usage
  - Similar to the VarNaming
  - Check all variables are placed appropriately

Agenda

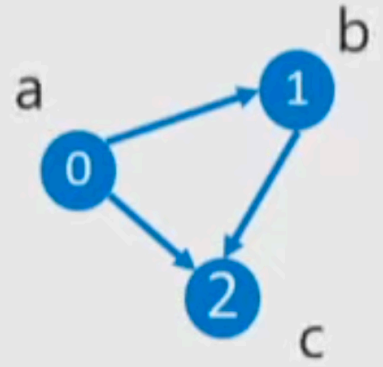
1. Problem / Related Work
2. Tasks
3. **Model**

Model

- Program Graphs
  - Syntax and semantic information
- Leveraging Variable Type Information
  - Map a variable's type to the set of its super type and choose maximum
  - Generalize to unseen types that implement common super types or interface
- Initial Node Representation
  - Split into subtoken, average, concatenate w/ type info, and pass it through a linear layer
- Gated Graph Neural Networks
- Program Graphs for VarNaming / VarMisuse
  - <SLOT> token
  - context and usage representation

Nodes & Edges

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathrm{N} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$



$$A \cdot N = \begin{bmatrix} 0 \\ a \\ a + b \end{bmatrix}$$

Cribbed from An Introduction to Graph Neural Networks: Models and Applications (https://youtu.be/zCEYiCxrL_0)

Syntax vs. Semantics

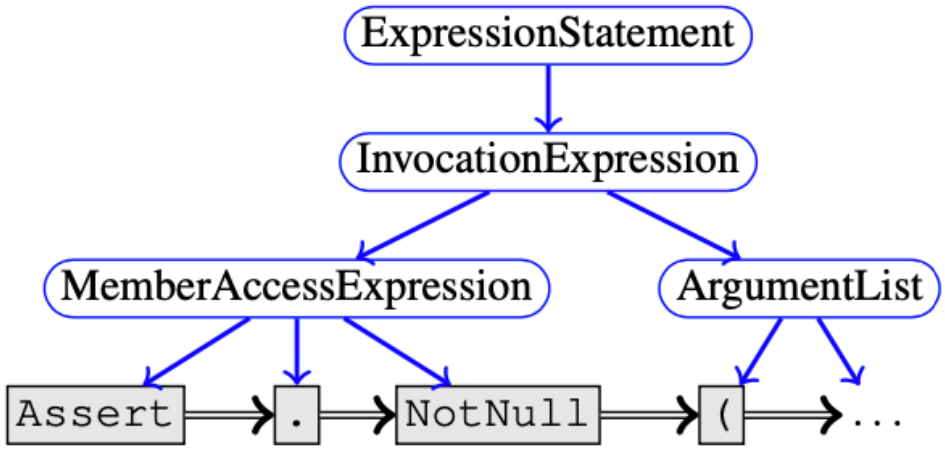
- Natural Language (Examples in CS682):
  - Take your money to the bank.
  - The river bank is muddy.
  - I wouldn't bank on it.

- Programming Language: (Objective Mini Pascal)

```
x.y
```

  - **`x`** is a record and **`y`** names a field of that record type
  - **`x`** refers to an object whose types has a field named **`y`**
  - **`x`** refers to an object whose type has a one-argument method named **`y`**

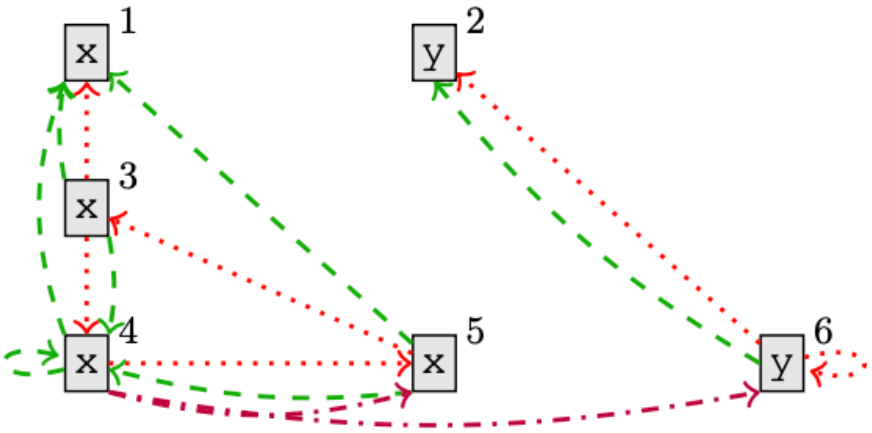
Syntax: NextToken & Child



(a) Simplified syntax graph for line 2 of Fig. 1, where blue rounded boxes are syntax nodes, black rectangular boxes syntax tokens, blue edges Child edges and double black edges NextToken edges.

Cribbed from Learning to Represent Programs with Graphs(https://arxiv.org/pdf/1711.00740)

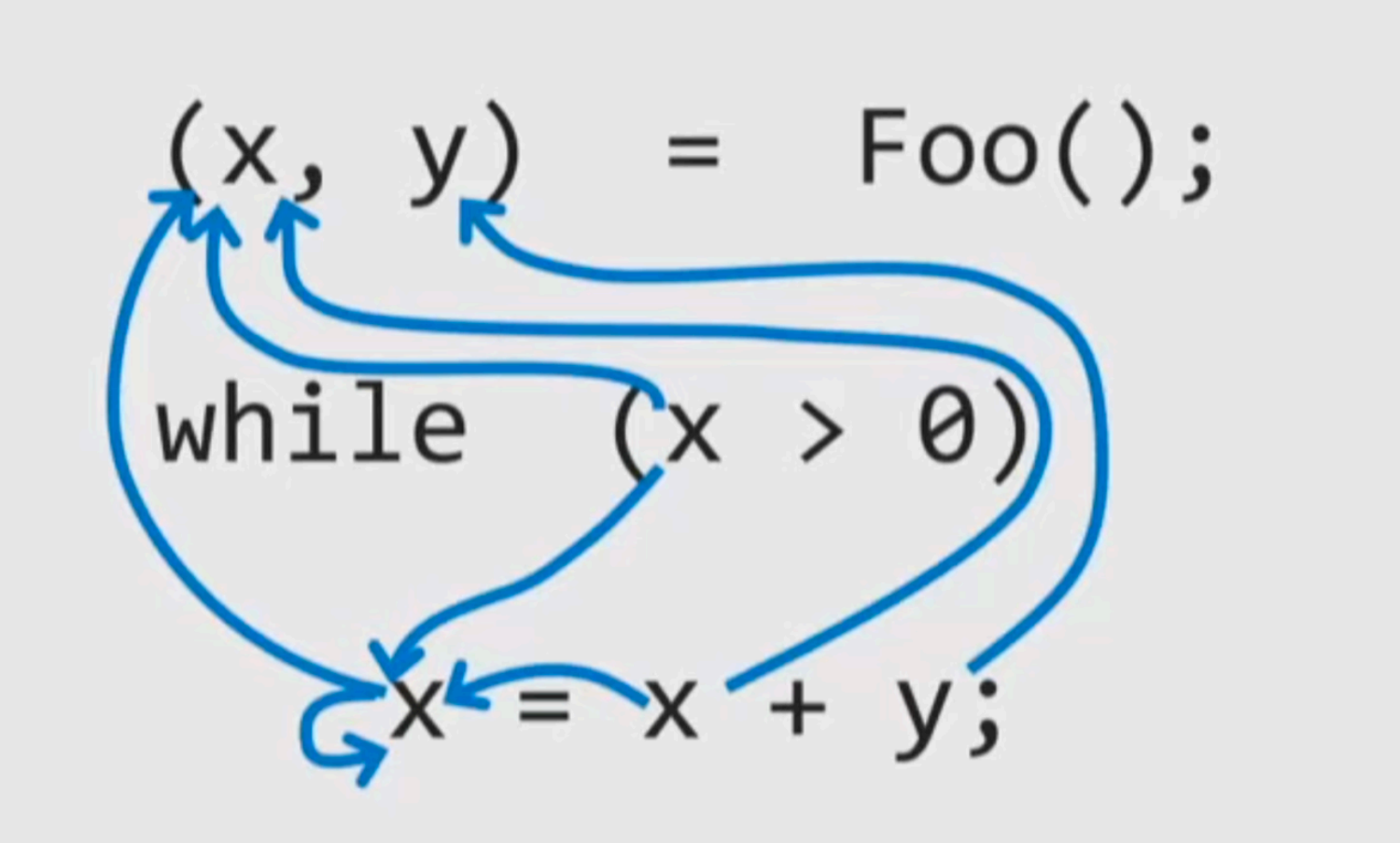
Semantics: Data Flow Edges



(b) Data flow edges for (x¹, y²) = Foo(); while (x³ > 0) x⁴ = x⁵ + y⁶ (indices added for clarity), with red dotted LastUse edges, green dashed LastWrite edges and dashdotted purple ComputedFrom edges.

Cribbed from Learning to Represent Programs with Graphs(https://arxiv.org/pdf/1711.00740)

Semantics: LastWrite

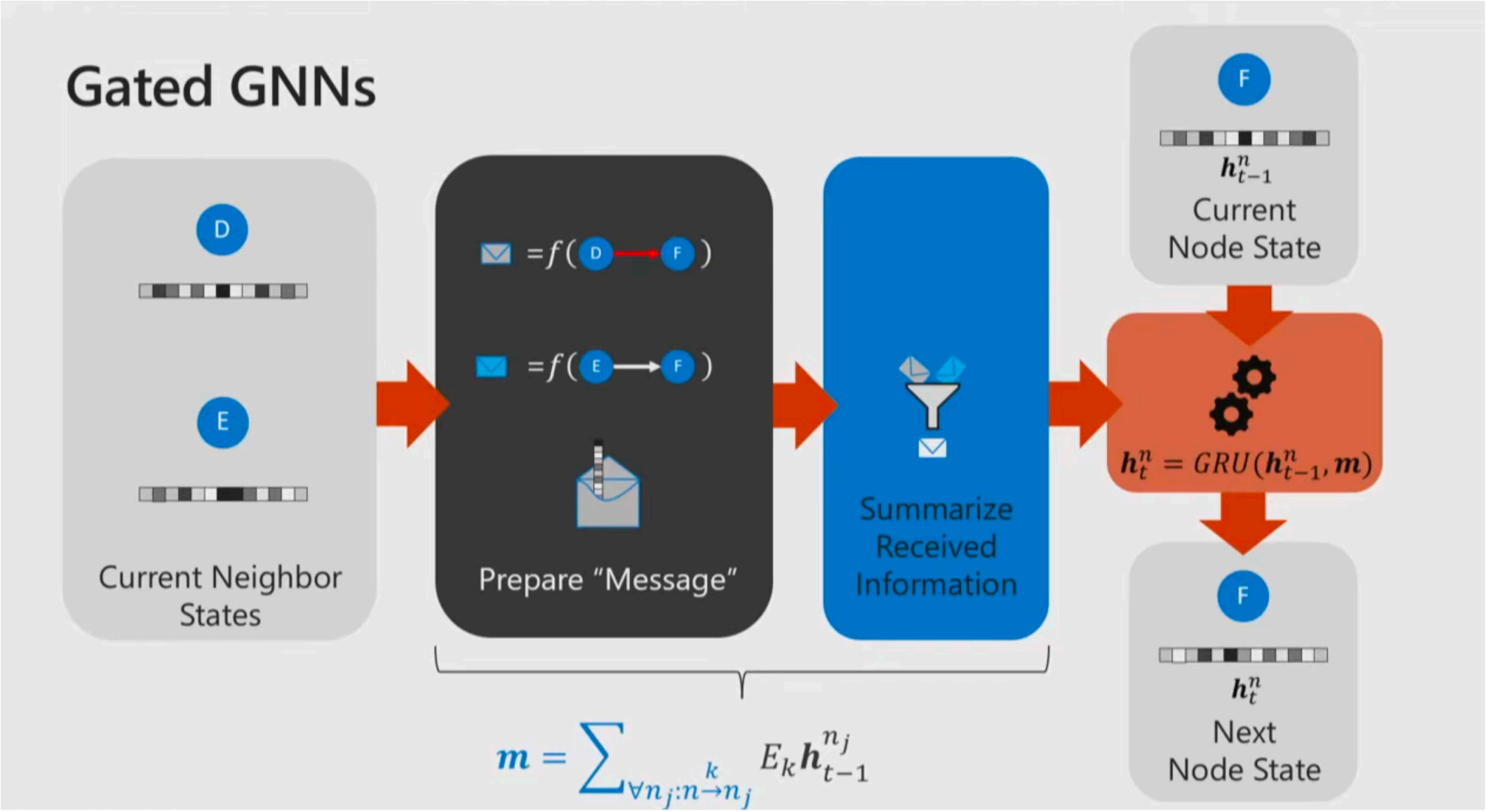


Cribbed from An Introduction to Graph Neural Networks: Models and Applications (https://youtu.be/zCEYiCxrL_0)

Edge Types in `varmisuse_task.py`

```
__PROGRAM_GRAPH_EDGES_TYPES =
["Child", "NextToken", "LastUse", "LastWrite", "LastLexicalUse",
"ComputedFrom", "GuardedByNegation", "GuardedBy", "FormalArgName",
"ReturnsTo", USES_SUBTOKEN_EDGE_NAME]
```

https://github.com/microsoft/tf-gnn-samples/blob/master/tasks/varmisuse_task.py



Cribbed from An Introduction to Graph Neural Networks: Models and Applications (https://youtu.be/zCEYiCxrL_0)

GGNN as Matrix Operation

Node States

$$H_t = \begin{bmatrix} \boldsymbol{h}_t^{n_0} \\ \vdots \\ \boldsymbol{h}_t^{n_K} \end{bmatrix} \quad \texttt{(num\_nodes x D)}$$

Messages to-be sent

$$M_t^k = E_k H_t \quad \texttt{(num\_nodes x M)}$$

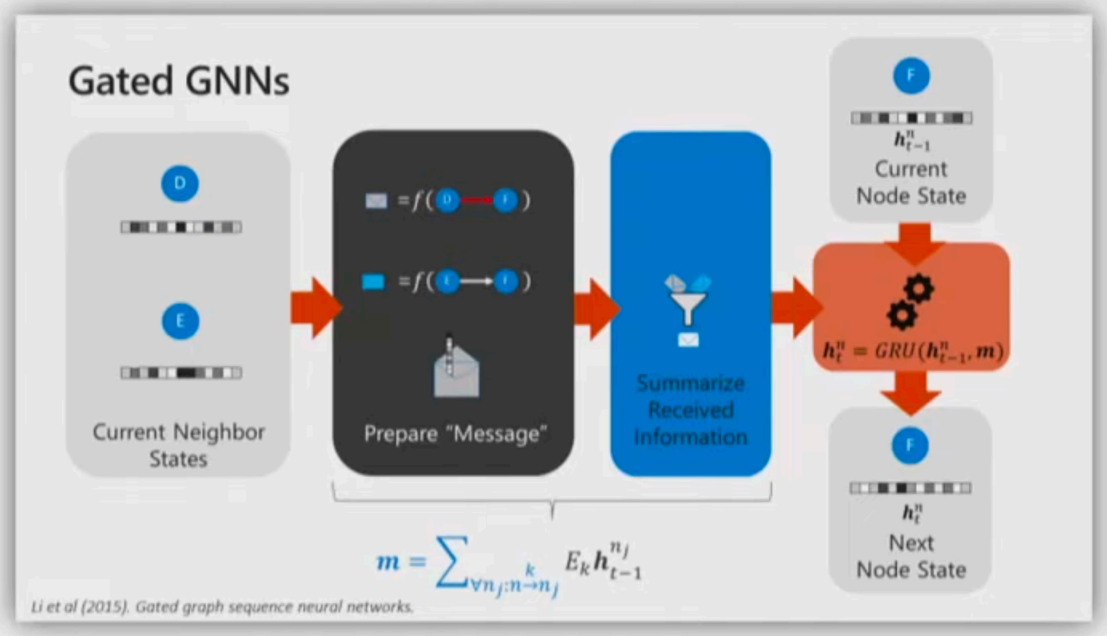
E: The type of the edge k

Received Messages

$$R_t = \sum_k A M_t^k \quad \texttt{(num\_nodes x M)}$$

A: The adjacency Matrix

Update $H_{t+1} = GRU(H_t, R_t)$



If we used a vanilla RNN

$$H_{t+1} = \sigma(\mathrm{U}H_t + \mathrm{W}R_t)$$

Cribbed from An Introduction to Graph Neural Networks: Models and Applications (https://youtu.be/zCEYiCxrL_0)

Agenda

1. Problem / Related Work
2. Tasks
3. Model
4. **Evaluation**

Name	Git SHA	kLOCs	Slots	Vars	Description
Akka.NET	719335a1	240	51.3k	51.2k	Actor-based Concurrent & Distributed Framework
AutoMapper	2ca7c2b5	46	3.7k	10.7k	Object-to-Object Mapping Library
BenchmarkDotNet	1670ca34	28	5.1k	6.1k	Benchmarking Library
BotBuilder	190117c3	44	6.4k	8.7k	SDK for Building Bots
choco	93985688	36	3.8k	5.2k	Windows Package Manager
commandline†	09677b16	11	1.1k	2.3k	Command Line Parser
CommonMark.NET^{Dev}	f3d54530	14	2.6k	1.4k	Markdown Parser
Dapper	931c700d	18	3.3k	4.7k	Object Mapper Library
EntityFramework	fa0b7ec8	263	33.4k	39.3k	Object-Relational Mapper
Hangfire	ffc4912f	33	3.6k	6.1k	Background Job Processing Library
Humanizer†	cc11a77e	27	2.4k	4.4k	String Manipulation and Formatting
Lean†	f574bfd7	190	26.4k	28.3k	Algorithmic Trading Engine
Nancy	72e1f614	70	7.5k	15.7	HTTP Service Framework
Newtonsoft.Json	6057d9b8	123	14.9k	16.1k	JSON Library
Ninject	7006297f	13	0.7k	2.1k	Code Injection Library
NLog	643e326a	75	8.3k	11.0k	Logging Library
Opserver	51b032e7	24	3.7k	4.5k	Monitoring System
OptiKey	7d35c718	34	6.1k	3.9k	Assistive On-Screen Keyboard
orleans	e0d6a150	300	30.7k	35.6k	Distributed Virtual Actor Model
Polly	0afdbc32	32	3.8k	9.1k	Resilience & Transient Fault Handling Library
quartznet	b33e6f86	49	9.6k	9.8k	Scheduler
ravendb^{Dev}	55230922	647	78.0k	82.7k	Document Database
RestSharp	70de357b	20	4.0k	4.5k	REST and HTTP API Client Library
Rx.NET	2d146fe5	180	14.0k	21.9k	Reactive Language Extensions
scriptcs	f3cc8bcb	18	2.7k	4.3k	C# Text Editor
ServiceStack	6d59da75	231	38.0k	46.2k	Web Framework
ShareX	718dd711	125	22.3k	18.1k	Sharing Application
SignalR	fa88089e	53	6.5k	10.5k	Push Notification Framework
Wox	cdaf6272	13	2.0k	2.1k	Application Launcher

Cribbed from Learning to Represent Programs with Graphs(https://arxiv.org/pdf/1711.00740.pdf)

Quantitative Evaluation

Table 1: Evaluation of models. SEENPROJTEST refers to the test set containing projects that have files in the training set, UNSEENPROJTEST refers to projects that have no files in the training data Results averaged over two runs.

	SEENPROJTEST				UNSEENPROJTEST			
	LOC	AVGLBL	AVGBIRNN	GGNN	LOC	AVGLBL	AVGBIRNN	GGNN
VARMISUSE								
Accuracy (%)	50.0	—	73.7	**85.5**	28.9	—	60.2	**78.2**
PR AUC	0.788	—	0.941	**0.980**	0.611	—	0.895	**0.958**
VARNAMING								
Accuracy (%)	—	36.1	42.9	**53.6**	—	22.7	23.4	**44.0**
F1 (%)	—	44.0	50.1	**65.8**	—	30.6	32.0	**62.0**

Cribbed from Learning to Represent Programs with Graphs(https://arxiv.org/pdf/1711.00740)

Quantitative Evaluation

Table 2: Ablation study for the GGNN model on SEENPROJTEST for the two tasks.

Ablation Description	Accuracy (%) VARMISUSE	VARNAMING
Standard Model (reported in Table 1)	85.5	53.6
Only NextToken, Child, LastUse, LastWrite edges	80.6	31.2
Only semantic edges (all but NextToken, Child)	78.4	52.9
Only syntax edges (NextToken, Child)	55.3	34.3
Node Labels: Tokens instead of subtokens	85.6	34.5
Node Labels: Disabled	84.3	31.8

Cribbed from Learning to Represent Programs with Graphs(https://arxiv.org/pdf/1711.00740)

Qualitative Evaluation

```
bool TryFindGlobalDirectivesFile(string baseDirectory, string fullPath, out string path){
  baseDirectory[1] = baseDirectory[2].TrimEnd(Path.DirectorySeparatorChar);
  var directivesDirectory = Path.GetDirectoryName(fullPath[3])
                                .TrimEnd(Path.DirectorySeparatorChar);
  while(directivesDirectory[4] != null && directivesDirectory[5].Length >= baseDirectory[6].Length){
    path[7] = Path.Combine(directivesDirectory[8], GlobalDirectivesFileName[9]);
    if (File.Exists(path[10])) return true;
    directivesDirectory[11]=Path.GetDirectoryName(directivesDirectory[12])
                                .TrimEnd(Path.DirectorySeparatorChar);
  }
  path[13] = null;
  return false;
}
```

1: path:59%, baseDirectory:35%, fullPath:6%, GlobalDirectivesFileName:1%
2: baseDirectory:92%, fullPath:5%, GlobalDirectivesFileName:2%, path:0.4%
3: fullPath:88%, baseDirectory:9%, GlobalDirectivesFileName:2%, path:1%
4: directivesDirectory:86%, path:8%, baseDirectory:2%, GlobalDirectivesFileName:1%, fullPath:0.1%
5: directivesDirectory:46%, path:24%, baseDirectory:16%, GlobalDirectivesFileName:10%, fullPath:3%
6: baseDirectory:64%, path:26%, directivesDirectory:5%, fullPath:2%, GlobalDirectivesFileName:2%
7: path:99%, directivesDirectory:1%, GlobalDirectivesFileName:0.5%, baseDirectory:7e-5, fullPath:4e-7
8: fullPath:60%, directivesDirectory:21%, baseDirectory:18%, path:1%, GlobalDirectivesFileName:4e-4
9: GlobalDirectivesFileName:61%, baseDirectory:26%, fullPath:8%, path:4%, directivesDirectory:0.5%
10: path:70%, directivesDirectory:17%, baseDirectory:10%, GlobalDirectivesFileName:1%, fullPath:0.6%
11: directivesDirectory:93%, path:5%, GlobalDirectivesFileName:1%, baseDirectory:0.1%, fullPath:4e-5%
12: directivesDirectory:65%, path:16%, baseDirectory:12%, fullPath:5%, GlobalDirectivesFileName:3%
13: path:97%, baseDirectory:2%, directivesDirectory:0.4%, fullPath:0.3%, GlobalDirectivesFileName:4e-4

Cribbed from Learning to Represent Programs with Graphs(https://arxiv.org/pdf/1711.00740)

Agenda

1. Problem / Related Work
2. Tasks
3. Model
4. Evaluation
5. **Limitation / Discussion**

Limitation / Discussion

- Backward Edges?
- Other Edge Types?
- Dynamic Typed Language?
- Comparison between static / runtime Analysis?

References

- Learning to Represent Programs with Graphs (https://arxiv.org/pdf/1711.00740)
- Gated Graph Sequence Neural Network(https://arxiv.org/pdf/1511.05493)
- Introduction to Graph Neural Networks: Models and Applications (https://youtu.be/zCEYiCxrL_0)

Thank you!

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

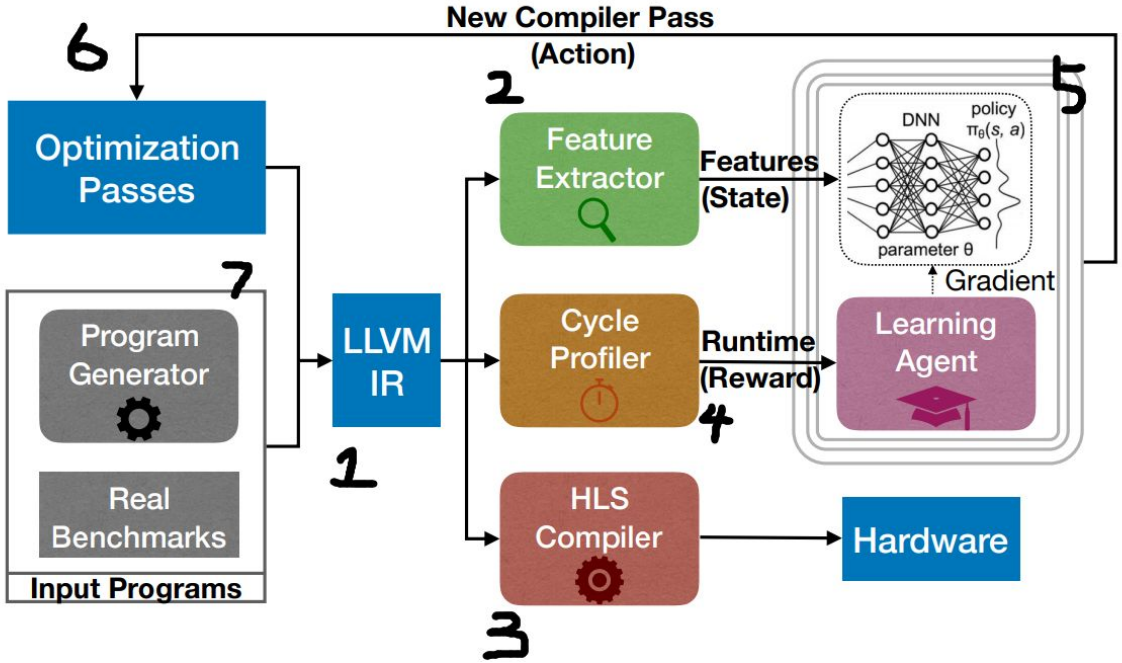
Evolutionary Algorithm

- 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

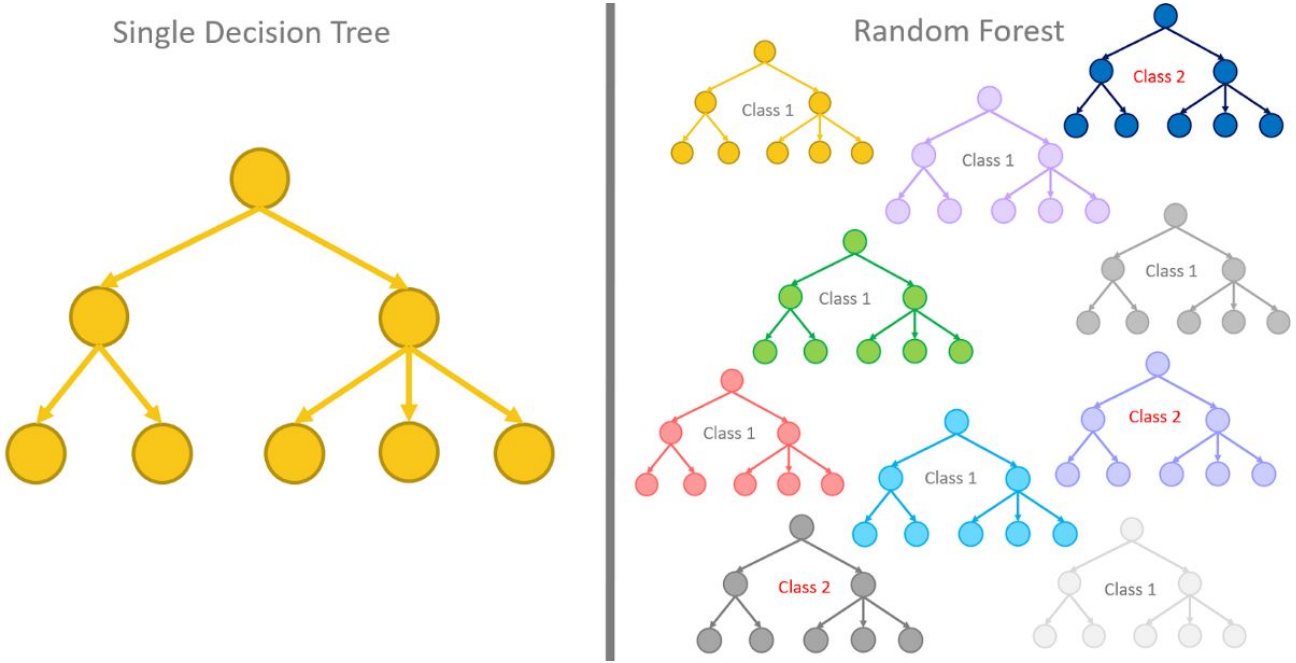
Reward: R = c(cycle count) curr - c previous

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

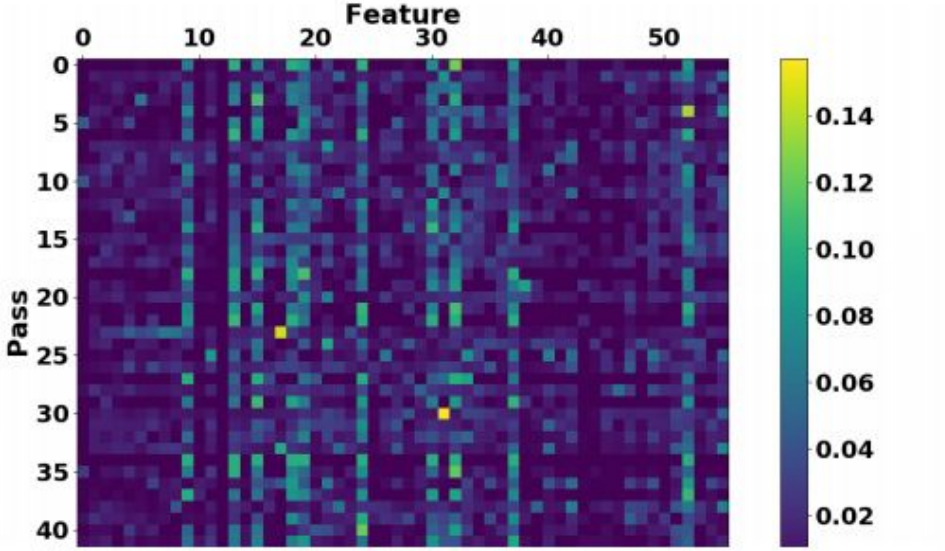


Figure 5: Heat map illustrating the importance of feature and pass indices.

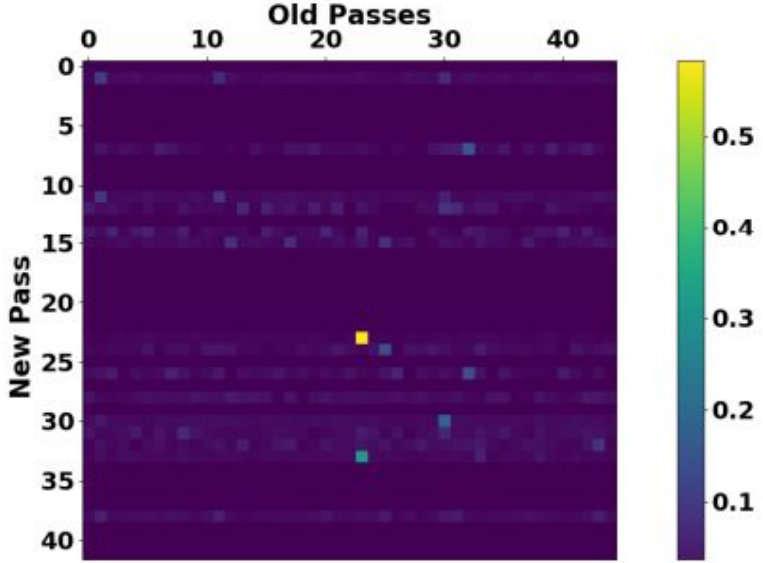


Figure 6: Heat map illustrating the importance of indices of previously applied passes and the new pass to apply.

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

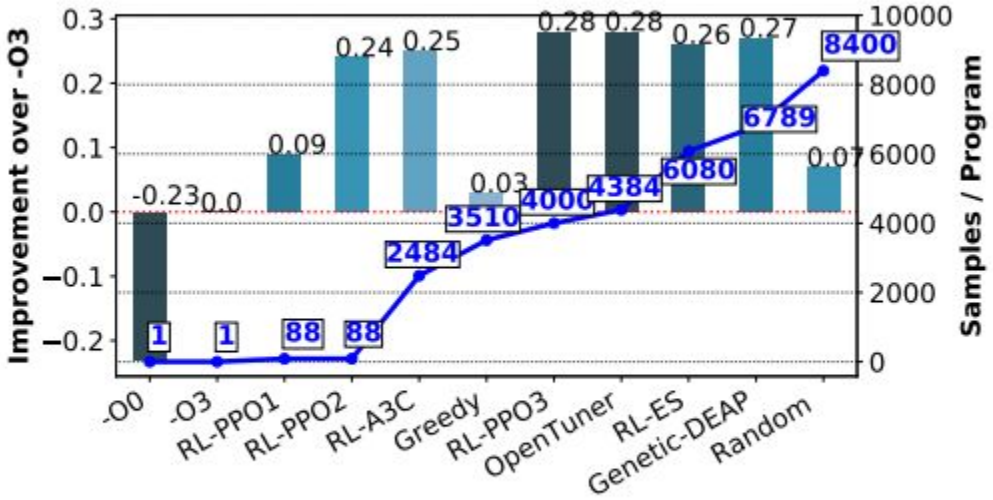


Figure 7: Circuit Speedup and Sample Size Comparison.

Generalization test using 100 Randomly generated programs

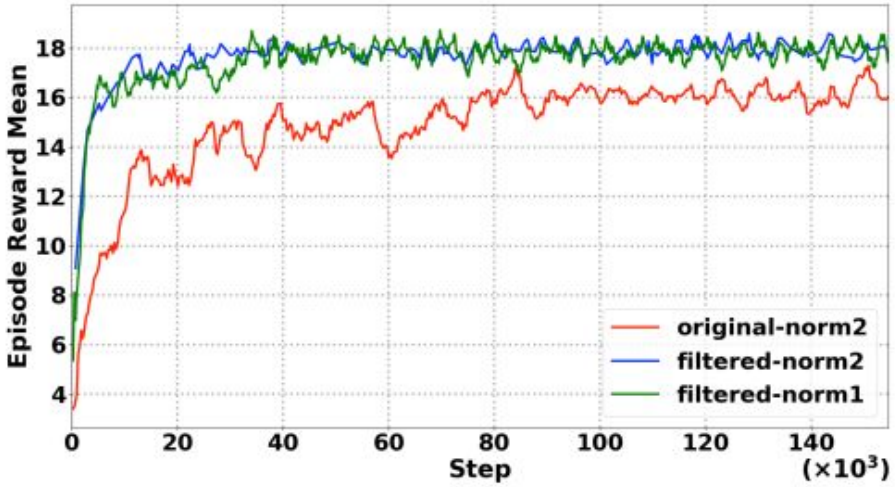


Figure 8: Episode reward mean as a function of step for the original approach where we use all the program features and passes and for the filtered approach where we filter the passes and features (with different normalization techniques). Higher values indicate faster circuit speed.

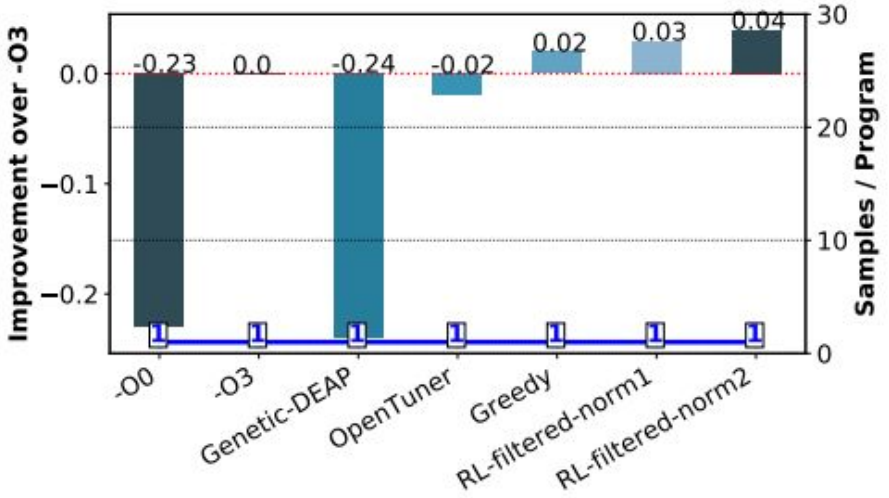


Figure 9: Circuit Speedup and Sample Size Comparison for deep RL Generalization.

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!

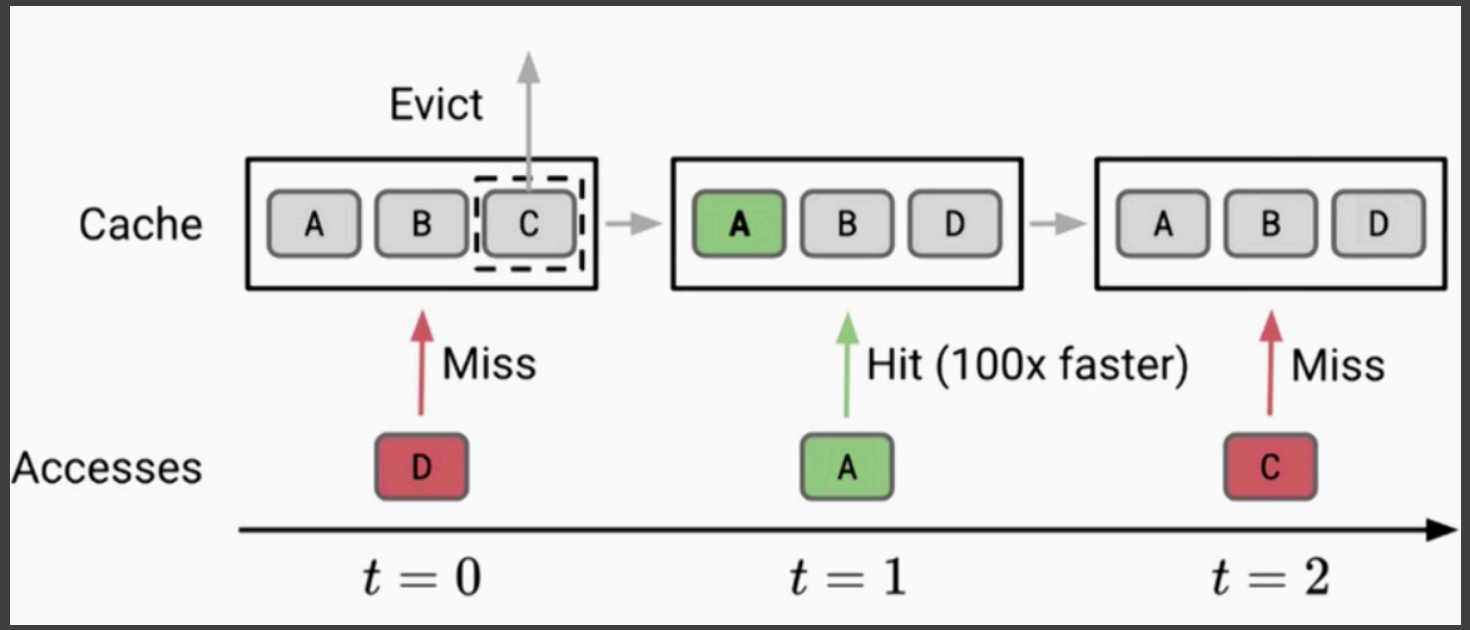
An Imitation Learning Approach for Cache Replacement

Jin Zhang

Cache

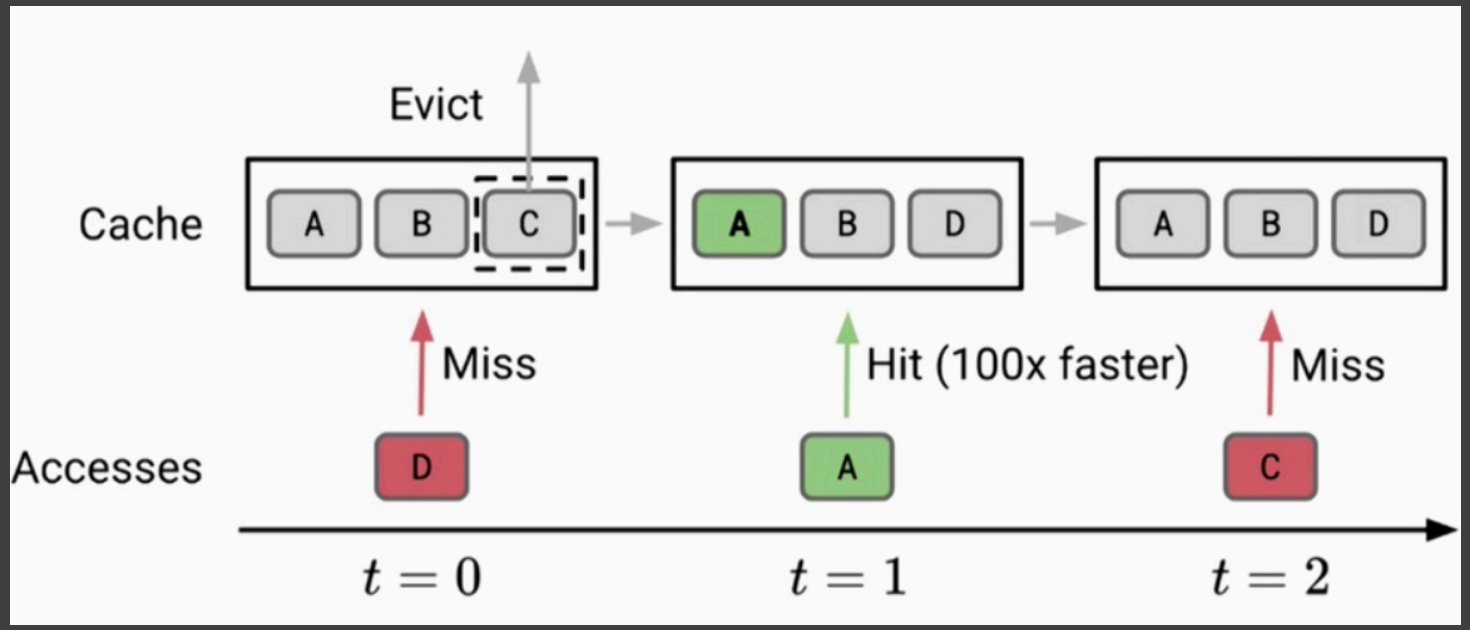
- Caching is a general concept in computer science found all over the software stack where data stored in different levels.
- Upper level: access faster, little data
- Lower level: access slower, lots of data
- Critical for latency: improving cache hit rates by just 1% can decrease total latency by as much as 35% for the web-scale applications ---Cidon et al. (2016)
- Cache replacement: strategically evicting data from the cache when making space for new data

Cache Replacement



- Single-level cache replacement
- Block of data = line
- Cache hit (fast): access to a block of data that is already in the cache. E.g. t = 1, access A
- Cache miss(slow): access to a block of data that is not in the cache. E.g. t = 0, access D; t = 2, access line C
- Cache replacement: when cache miss, an existing cache line must be evicted from the cache to make space for the new line

Cache Replacement



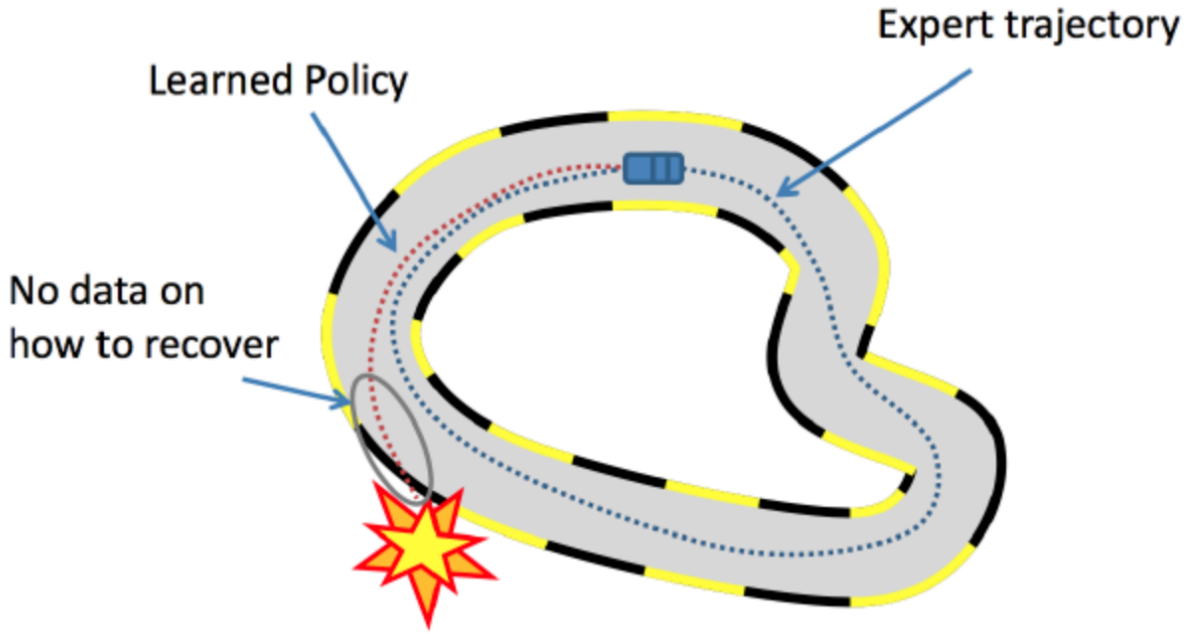
- Reuse distance dt(lw): number of total cache accesses until next access to lw. E.g. d0(A)=1, d0(B)>2, d0(C)=2
- Belady's optimal policy : given future cache accesses, evict the line with the highest reuse distance
- Problem: require future information
- Heuristics in practice: most recently used, least recently used, frequently reused vs not---poor at complex pattens

Outline

- Cast cache replacement as an imitation learning problem, leveraging Belady's in a new way called PARROT, conditioned only with past access.
- Establish a neural architecture to realize the end-to-end cache replacement.
- PARROT outperforms the state-of- the-art replacement policy's hit rates by over 20% on memory-intensive CPU benchmarks.
- PARROT improves cache hit rates by 61% over a commonly implemented LRU policy on an industrial-scale web search workload

Imitation learning

- Imitation learning techniques aim to mimic human behavior in a given task.
- An agent (a learning machine) is trained to perform a task from demonstrations by learning a mapping between observations and actions



Casting Cache Replacement as Imitation Learning

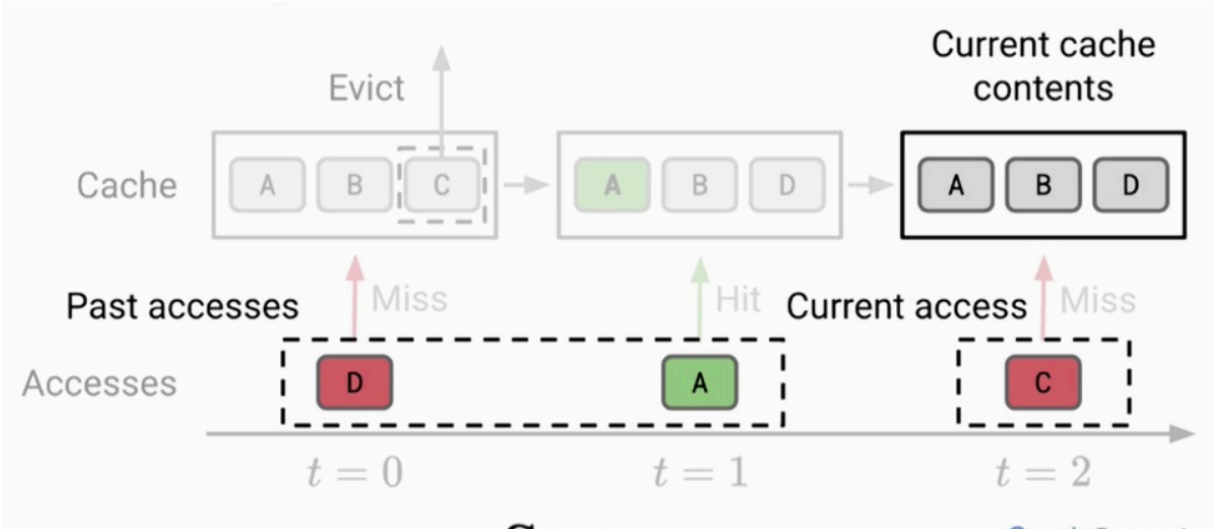
- Markov decision process ⟨S , A , R, P ⟩
- State S: $s_t = (s_t^c, s_t^a, s_t^h)$
- $s_t^a = (m_t, pc_t)$ = (currently accessed cache line address, unique program counter of the access)
- $s_t^c = \{l_1, \ldots, l_W\}$ = cache state consisting of W cache lines
- $s_t^h = (\{m_1, \ldots, m_{t-1}\}, \{pc_1, \ldots, pc_{t-1}\})$ = history of all past cache accesses
- Action A: evict line {1,2….W} / no-op
- Reward R: R(cache hit)=1, R(cache miss)=0
- Transition: $P(s_{t+1} \mid a_t, s_t)$

$$s_{t+1}^a = (m_{t+1}, pc_{t+1}) \qquad s_{t+1}^h = (\{m_1, \ldots, m_{t-1}, m_t\}, \{pc_1, \ldots, pc_{t-1}, pc_t\}).$$

$$s_{t+1}^c = \{l_1, \ldots, l_{w-1}, l_{w+1}, \ldots, l_W, m_t\} \text{ where } a_t = w.$$

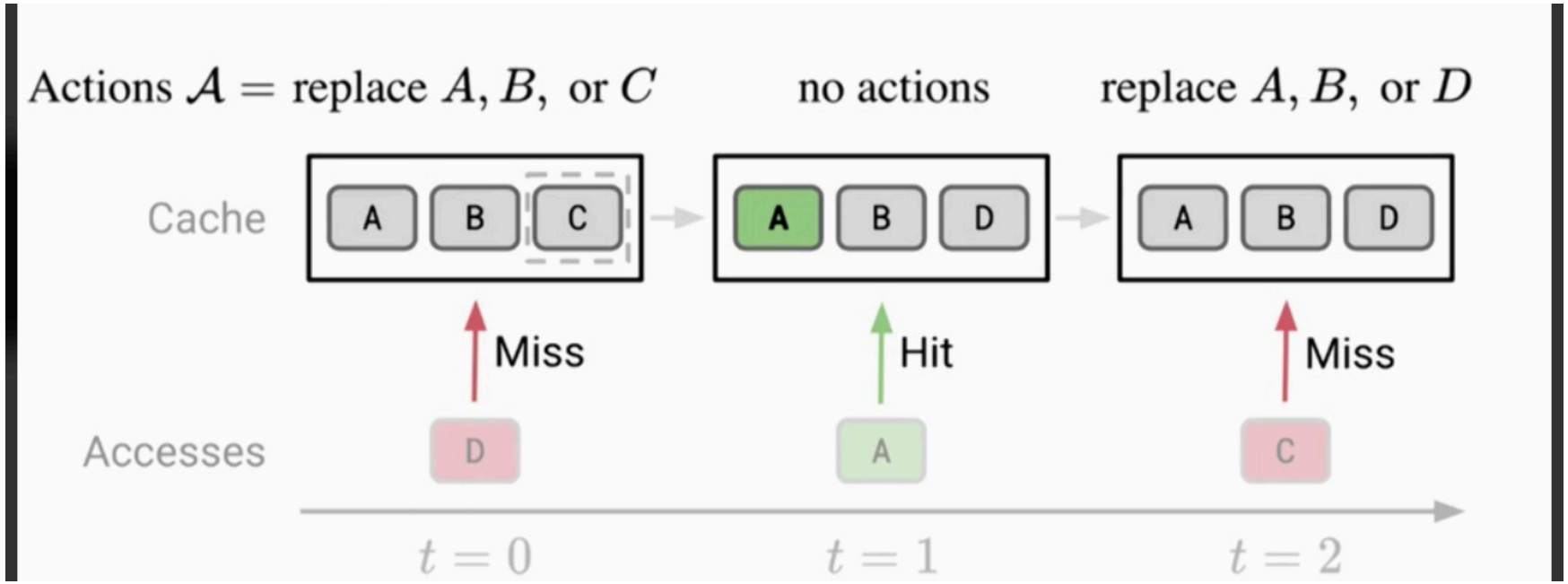
Casting Cache Replacement as Imitation Learning

- Markov decision process ⟨S , A , R, P ⟩
- State S: $s_t = (s_t^c, s_t^a, s_t^h)$
- $s_t^a = (m_t, pc_t)$ = (currently accessed cache line address, unique program counter of the access)
- $s_t^c = \{l_1, \ldots, l_W\}$ = cache state consisting of W cache line
- $s_t^h = (\{m_1, \ldots, m_{t-1}\}, \{pc_1, \ldots, pc_{t-1}\})$ = history of all past cache accesses



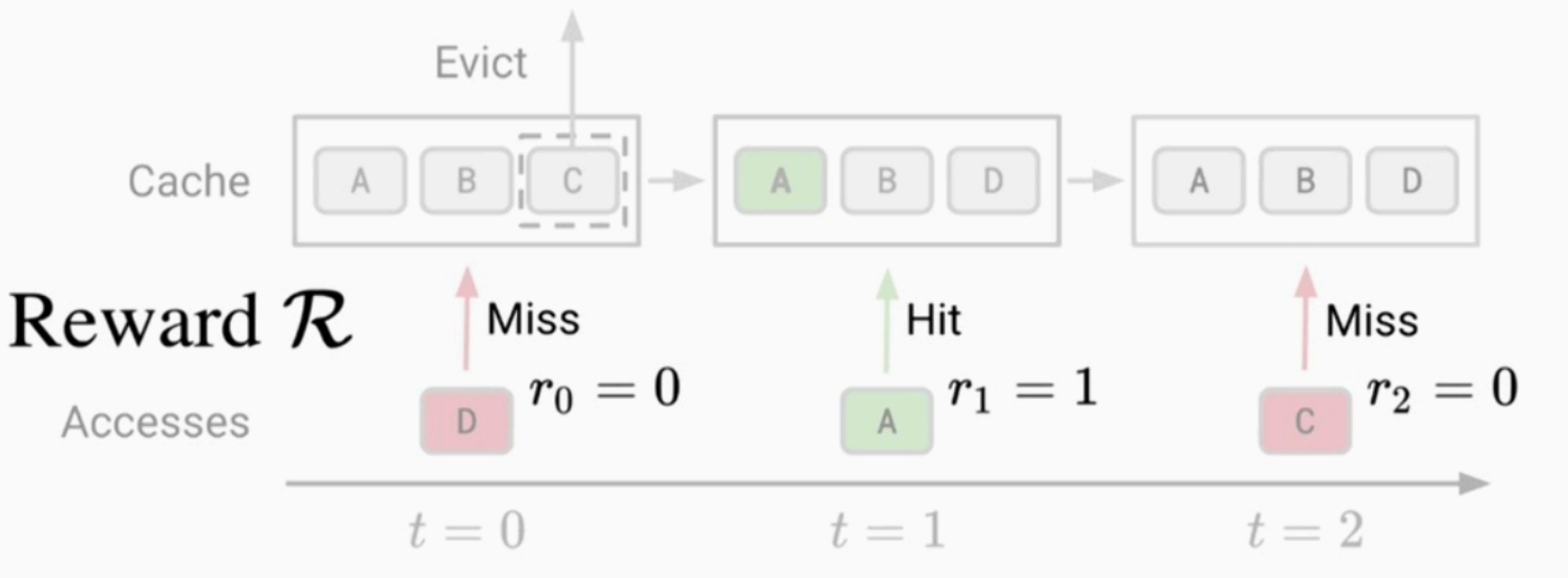
Casting Cache Replacement as Imitation Learning

- Markov decision process ⟨S , A , R, P ⟩
- Action A: evict line {1,2….W} / no-op



Casting Cache Replacement as Imitation Learning

- Markov decision process ⟨S , A , R, P ⟩
- Reward R: R(cache hit)=1, R(cache miss)=0



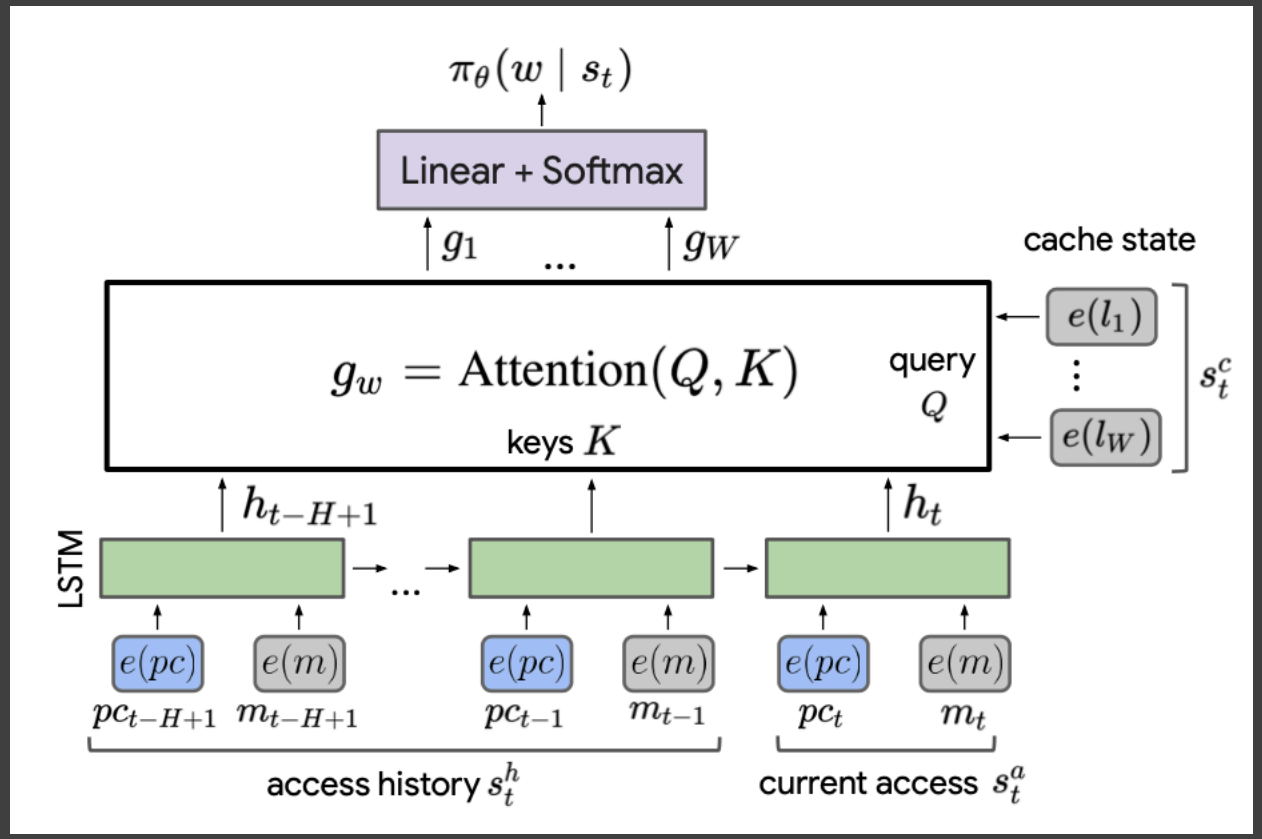
Casting Cache Replacement as Imitation Learning

- Markov decision process ⟨S , A , R, P ⟩
- State S: $s_t = (s_t^c, s_t^a, s_t^h)$
- $s_t^a = (m_t, pc_t)$ = (currently accessed cache line address, unique program counter of the access)
- $s_t^c = \{l_1, \ldots, l_W\}$ = cache state consisting of W cache line
- $s_t^h = (\{m_1, \ldots, m_{t-1}\}, \{pc_1, \ldots, pc_{t-1}\})$ = history of all past cache accesses
- Action A: evict line {1,2….W} / no-op
- Reward R: R(cache hit)=1, R(cache miss)=0
- Transition: $P(s_{t+1} \mid a_t, s_t)$

$$s_{t+1}^a = (m_{t+1}, pc_{t+1}) \qquad s_{t+1}^h = (\{m_1, \ldots, m_{t-1}, m_t\}, \{pc_1, \ldots, pc_{t-1}, pc_t\}).$$

$$s_{t+1}^c = \{l_1, \ldots, l_{w-1}, l_{w+1}, \ldots, l_W, m_t\} \text{ where } a_t = w.$$

Neural architecture



1. Embed the current cache access $s_t^a = (m_t, pc_t)$ to obtain memory address embedding $e(m_t)$ and PC embedding $e(pc_t)$ and pass them through an LSTM to obtain cell state c_t and hidden state h_t:

$$c_t, h_t = \text{LSTM}([e(m_t); e(pc_t)], c_{t-1}, h_{t-1})$$

2. Keep the past H hidden states, $[h_{t-H+1}, \ldots, h_t]$, representing an embedding of the cache access history s_t^h and current cache access s_t^a.
3. Form a context g_w for each cache line l_w in the cache state s_t^c by embedding each line as $e(l_w)$ and attending over the past H hidden states with $e(l_w)$:

$$g_w = \text{Attention}(Q, K)$$

where query $Q = e(l_w)$, keys $K = [h_{t-H+1}, \ldots, h_t]$

4. Apply a final dense layer and softmax on top of these line contexts to obtain the policy:

$$\pi_\theta(a_t = w \mid s_t) = \text{softmax}(\text{dense}(g_w))$$

5. Choose $\arg\max_{a \in \mathcal{A}_{s_t}} \pi_\theta(a \mid s_t)$ as the replacement action to take at timestep t.

Training Algorithm for PARROT policy πθ

Algorithm 1 PARROT training algorithm

1: Initialize policy π_θ
2: **for** step $= 0$ **to** K **do**
3: **if** step $\equiv 0 \pmod{5000}$ **then**
4: Collect data set of visited states $B = \{s_t\}_{t=0}^{T}$ by following π_θ on all accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$
5: **end if**
6: Sample contiguous accesses $\{s_t\}_{t=l-H}^{l+H}$ from B
7: Warm up policy π_θ on initial H accesses $(m_{l-H}, pc_{l-H}), \ldots, (m_l, pc_l)$
8: Compute loss $\mathcal{L} = \sum_{t=l}^{l+H} \mathcal{L}_\theta(s_t, \pi^*)$
9: Update policy parameters θ based on loss $\mathcal{L}$
10: **end for**

Step 3-5

- DAgger algorithm in training
- Following current learned policy πθ as in the testing, instead of oracle policy π*
- Avoid compounding errors by keeping the distribution of training states and test states the same.

Training Algorithm for PARROT policy πθ

Algorithm 1 PARROT training algorithm

1: Initialize policy π_θ
2: **for** step $= 0$ **to** K **do**
3: **if** step $\equiv 0 \pmod{5000}$ **then**
4: Collect data set of visited states $B = \{s_t\}_{t=0}^{T}$ by following π_θ on all accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$
5: **end if**
6: Sample contiguous accesses $\{s_t\}_{t=l-H}^{l+H}$ from B
7: Warm up policy π_θ on initial H accesses $(m_{l-H}, pc_{l-H}), \ldots, (m_l, pc_l)$
8: Compute loss $\mathcal{L} = \sum_{t=l}^{l+H} \mathcal{L}_\theta(s_t, \pi^*)$
9: Update policy parameters θ based on loss $\mathcal{L}$
10: **end for**

Step 6-9

- updates the PARROT policy to approximate Belady's policy by minizine the loss

Training Algorithm for PARROT policy πθ

Algorithm 1 PARROT training algorithm

1: Initialize policy π_θ
2: **for** step $= 0$ **to** K **do**
3: **if** step $\equiv 0 \pmod{5000}$ **then**
4: Collect data set of visited states $B = \{s_t\}_{t=0}^{T}$ by following π_θ on all accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$
5: **end if**
6: Sample contiguous accesses $\{s_t\}_{t=l-H}^{l+H}$ from B
7: Warm up policy π_θ on initial H accesses $(m_{l-H}, pc_{l-H}), \ldots, (m_l, pc_l)$
8: Compute loss $\mathcal{L} = \sum_{t=l}^{l+H} \mathcal{L}_\theta(s_t, \pi^*)$
9: Update policy parameters θ based on loss $\mathcal{L}$
10: **end for**

Step 8

- Observation: not all errors are equally bad
- Replace the simple log-likelihood behavior cloning loss function
- With an alternate ranking loss function
- Penalizing heavily on cache miss while lightly on cache hit

Training Algorithm for PARROT policy πθ

Algorithm 1 PARROT training algorithm

1: Initialize policy π_θ
2: **for** step $= 0$ **to** K **do**
3: **if** step $\equiv 0 \pmod{5000}$ **then**
4: Collect data set of visited states $B = \{s_t\}_{t=0}^{T}$ by following π_θ on all accesses $(m_1, pc_1), \ldots, (m_T, pc_T)$
5: **end if**
6: Sample contiguous accesses $\{s_t\}_{t=l-H}^{l+H}$ from B
7: Warm up policy π_θ on initial H accesses $(m_{l-H}, pc_{l-H}), \ldots, (m_l, pc_l)$
8: Compute loss $\mathcal{L} = \sum_{t=l}^{l+H} \mathcal{L}_\theta(s_t, \pi^*)$
9: Update policy parameters θ based on loss $\mathcal{L}$
10: **end for**

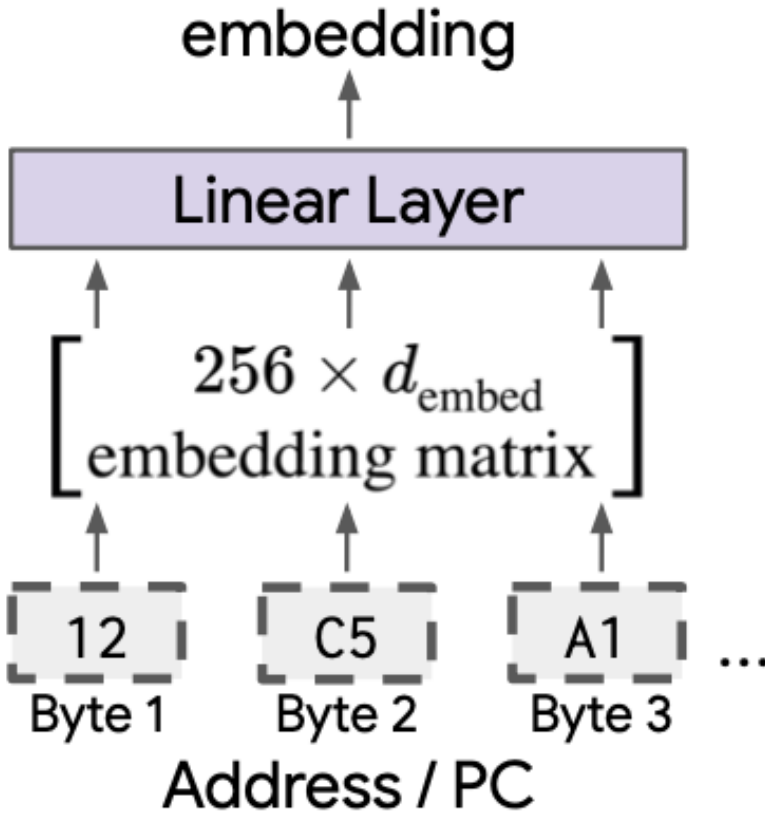
Step 8

- Observation: predict reuse distance is correlated with cache replacement.
- Add reuse distance predict loss

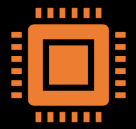
$$\mathcal{L}_\theta(s, \pi^*) = \mathcal{L}_\theta^{\text{rank}}(s, \pi^*) + \mathcal{L}_\theta^{\text{reuse}}(s, \pi^*)$$

Byte Embedder

- Full-sized PARROT model learn a separate embedding for each PC and memory address
- To reduce model size, byte embedder shared across all memory addresses, only requiring several kilobytes of storage. This byte embedder embeds each memory address (or PC) by embedding each byte separately and then passing a small linear layer over their concatenated outputs



Experiments---Setup

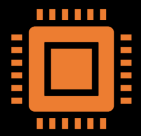


Evaluated on 3-level cache hierarchy with a 4-way 32 KB L1 cache and 1 8-way 256 KB L2 cache, which are based on LRU policy, and then 1 16-way 2 MB L3 cache, which is based on PARROT policy.

Collect raw memory access traces over 50s interval using dynamic binary instrumentation tools and then randomly choose 64 sets and collect accesses to them on L3 cache, with an average of 5M accesses in total per program. 80% for training, 10% for validation and 10% for testing.

Two evaluation metrics: (1) raw cache hit rates (2) normalized cache hit rates = (r−rLRU) /(ropt −rLRU), rLRU and ropt are the hit ratesof LRU and Belady’s,

Experiments---Benchmark



Memory-intensive SPEC2006 CPU applications

Industrial-scale application Google Web Search

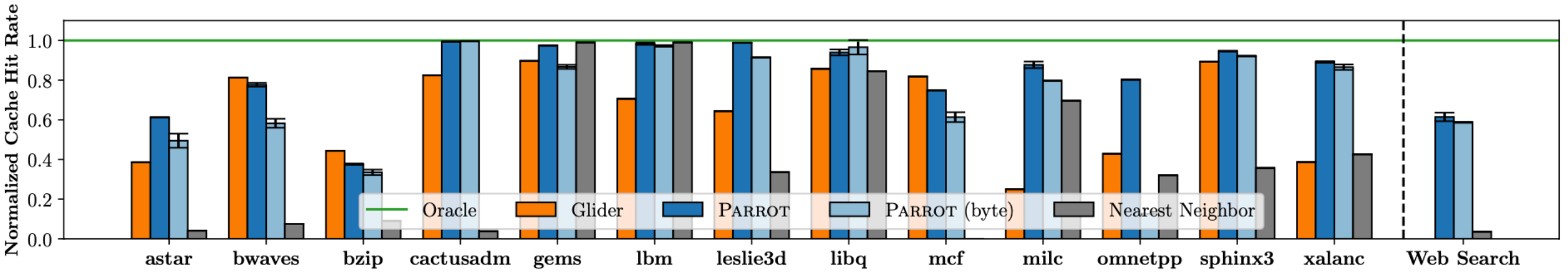
Experiments Results

- Comparison of raw cache hit rate between PARROT, Belady's and LRU on SPEC2006 applications
- Significantly higher cache hit rates in PARROT than LRU on every program [2%~ 30%, average 16%]
- For Google Web Search: 61% higher normalized cache hit rate and 13.5% higher raw cache hit rate than LRU

	astar	bwaves	bzip	cactusadm	gems	lbm	leslie3d	libq	mcf	milc	omnetpp	sphinx3	xalanc	Web Search
Optimal	43.5%	8.7%	78.4%	38.8%	26.5%	31.3%	31.9%	5.8%	46.8%	2.4%	45.1%	38.2%	33.3%	67.5%
LRU	20.0%	4.5%	56.1%	7.4%	9.9%	0.0%	12.7%	0.0%	25.3%	0.1%	26.1%	9.5%	6.6%	45.5%
PARROT	34.4%	7.8%	64.5%	38.6%	26.0%	30.8%	31.7%	5.4%	41.4%	2.1%	41.4%	36.7%	30.4%	59.0%

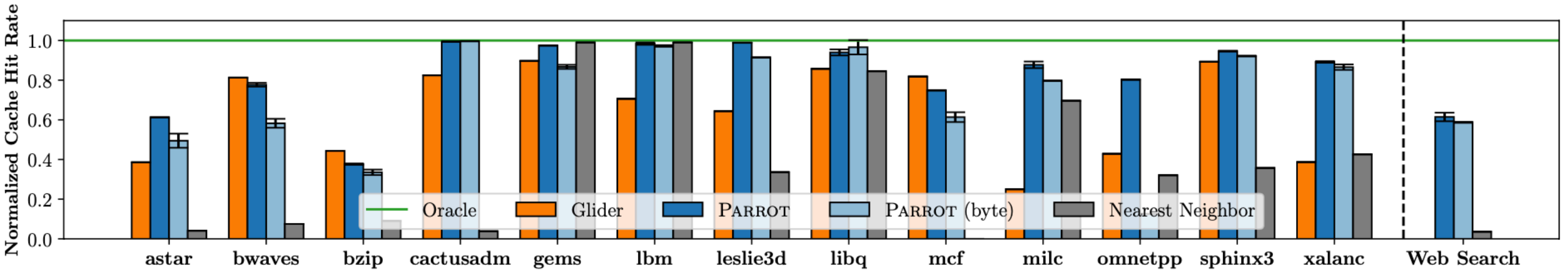
Experiments Results

- Comparison of normalized cache hit rates between full-sized PARROT, byte embedder model PARROT(byte), state-of-the-art replacement policy Glider and Nearest Neighbor Belady's on SPEC2006 applications
- PARROT outperforms Glider on 10 of the 13 SPEC2006 programs, achieving an average of 20% higher normalized cache hit rate averaged over all programs



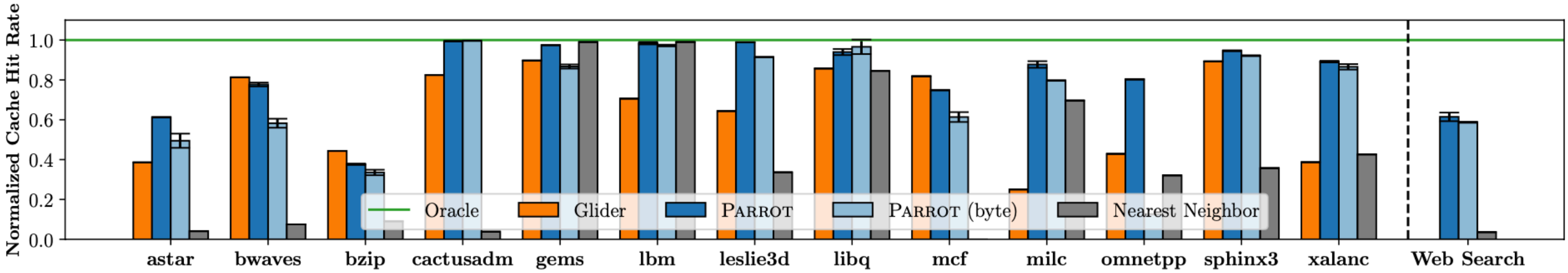
Experiments Results

- Comparison of normalized cache hit rates between full-sized PARROT, byte embedder model PARROT(byte), state-of-the-art replacement policy Glider and Nearest Neighbor Belady's on SPEC2006 applications
- PARROT(byte) is smaller than PARROT, but it still achieves an average of 8% higher normalized cache hit rate than Glider



Experiments Results

- Comparison of normalized cache hit rates between full-sized PARROT, byte embedder model PARROT(byte), state-of-the-art replacement policy Glider and Nearest Neighbor Belady's on SPEC2006 applications
- PARROT maintains high normalized cache hit rates on both simple and complex programs. Nearest Neighbor Belady's performs good only on simple programs (e.g., gems, lbm) but it fails for more complex programs (e.g., mcf, Web Search)



Experiments Results

- Experiment with each fuction component, e.g. predicting reuse distance, on-policy training (DAgger) and ranking loss.
- PARROT (no reuse distance)
- PARROT(evict highest reuse distance)
- PARROT (base): predict reuse distance and directly evicts line with the highest predicted reuse distance
- PARROT (reuse distance aux loss) predict reuse distance as an auxiliary task
- Some programs, e.g.omnetpp and lbm, simple PARROT neural architecture is sufficient while in other programs, additional component are required to achieve state-of-the-art cache hit rates.

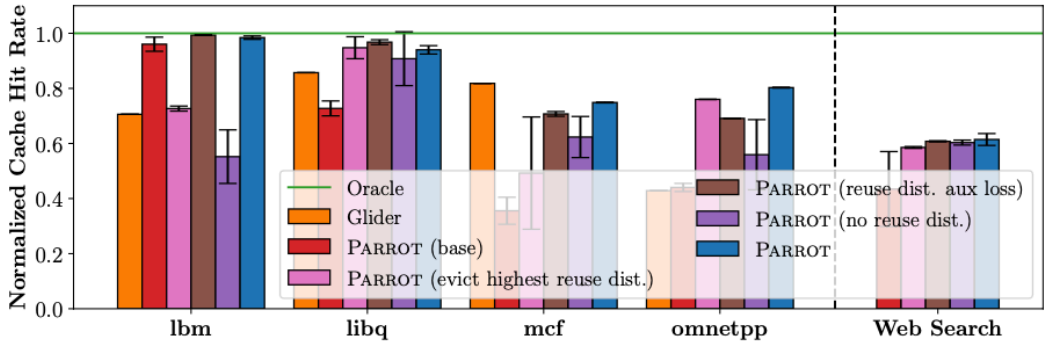


Figure 6. Comparison between different mechanisms of incorporating reuse distance into PARROT. Including reuse distance prediction in our full model (PARROT) achieves 16.8% higher normalized cache hit rates than ablating reuse distance prediction (PARROT (no reuse dist.)).

Experiments Results

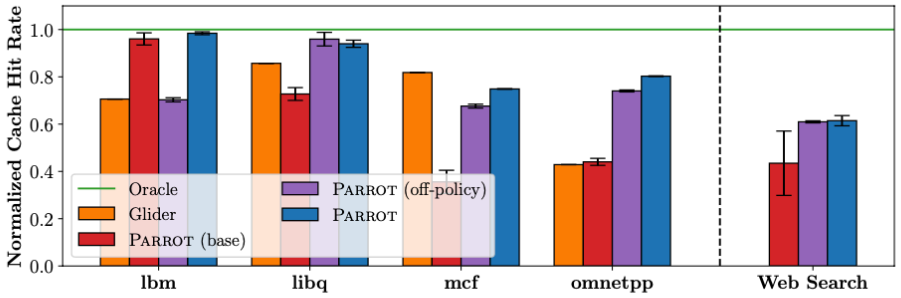


Figure 7. Ablation study for training with DAgger. Training with DAgger achieves 9.8% higher normalized cache hit rates than training off-policy on the states visited by the oracle policy.

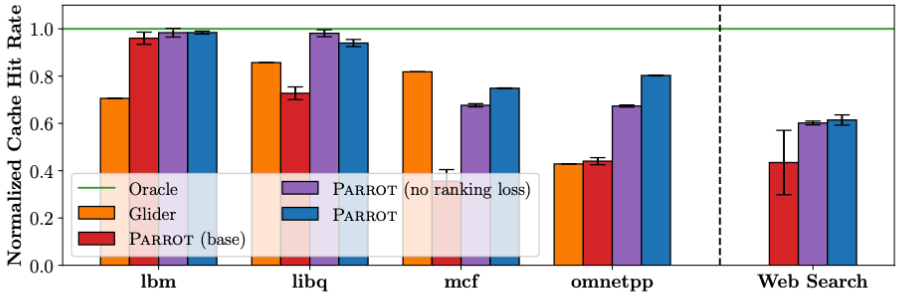


Figure 8. Ablation study for our ranking loss. Using our ranking loss improves normalized cache hit rate by 3.5% over a LL loss.

- Experiment with on-policy training (DAgger)
- In theory, training off-policy should lead to compounding errors
- Empirically, this is highly program-dependent.
- E.g. in mcf or Web Search, training off-policy performs as well or better than training on-policy
- In other programs, training on-policy is crucial.
- Over-all, training on-policy leads to an average of 9.8% normalized cache hit rate improvement over off-policy training.

- Ranking loss always provides a greater performance.

Experiments Results

Experiment with the number of past accesses that PARROT attends

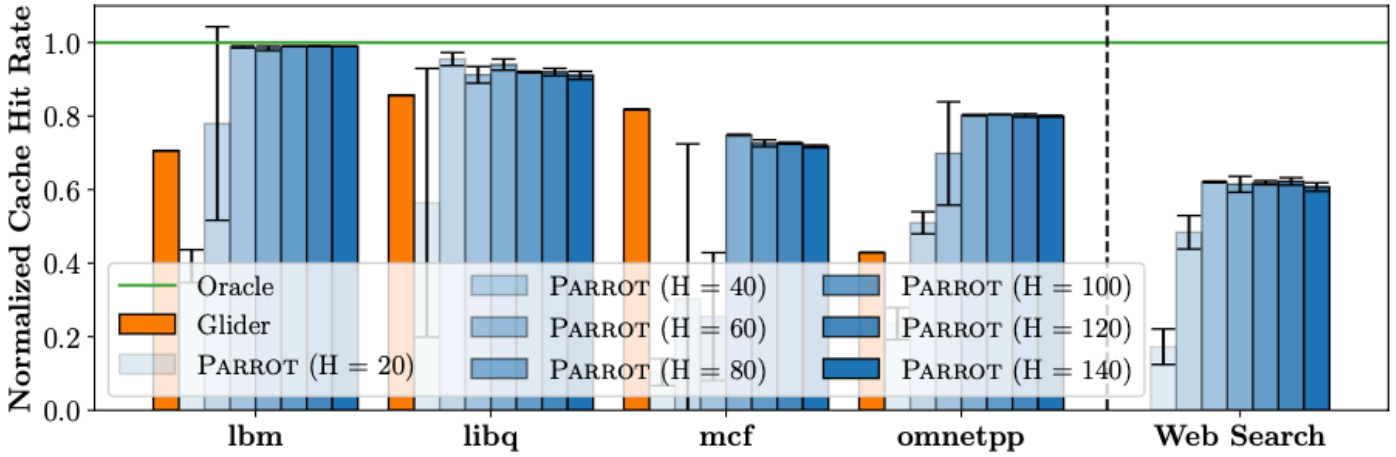


Figure 9. Performance of PARROT trained with different numbers of past accesses (H). As the number of past accesses increases, normalized cache hit rates improve, until reaching a history length of 80. At that point, additional past accesses have little impact.

Perspective

- The first method deal with cache replacement based on imitation learning.
- The performance is actually good.
- Memory and latency overheads of the policy are the important concerns need to be considered. However, the paper does not show the influence for the PARROT on them.
- It only works on the single-level cache, there is no known for multiple levels of caches (as is common in CPUs and web services).

Thanks

Jin Zhang