# Machine Learning for Software System / Hardware Design: Towards AI-Assisted Programming Tasks

**Hanxian Huang**
**(hah008@ucsd.edu, https://hanxian97.github.io)**
**Advisor: Prof. Jishen Zhao**

## Towards LLM-Powered Verilog RTL Assistant: Self-Verification and Self-Correction

*Hanxian Huang[1], Zhenghan Lin[2], Zixuan Wang[1], Xin Chen[3], Ke Ding[3], Jishen Zhao[1]*
*University of California San Diego[1], University of California Berkeley[2], Applied ML Group Intel Corp.[3]*
* Will be presented at HotChips 2024 tutorial

**Background and Motivation:**

- Complexity of Traditional RTL Design
  - Describe architectures and behaviors at a granular level
- Differences between HDLs and General-Purpose PLs
  - RTL design is more complex considering timing constraints
  - RTL verification is hard considering efficiency and coverage
  - Existing code-LLMs are not tailored for RTL design

**Methodology:**

- Leverage LLMs code generation ability, iterative interaction ability, and Chain-of-thought ability
- Design prompts by mimicking human designers behavior:
  - Reason and solve the design problem step by step
  - Generate testbench with test cases, and walk through code to deductively reason the code behavior considering timing, given a certain input or a previously failed input test case
  - Based on the simulator feedback and code walk-through process, revise code, fix bugs, and meet design specifications over multiple iterations

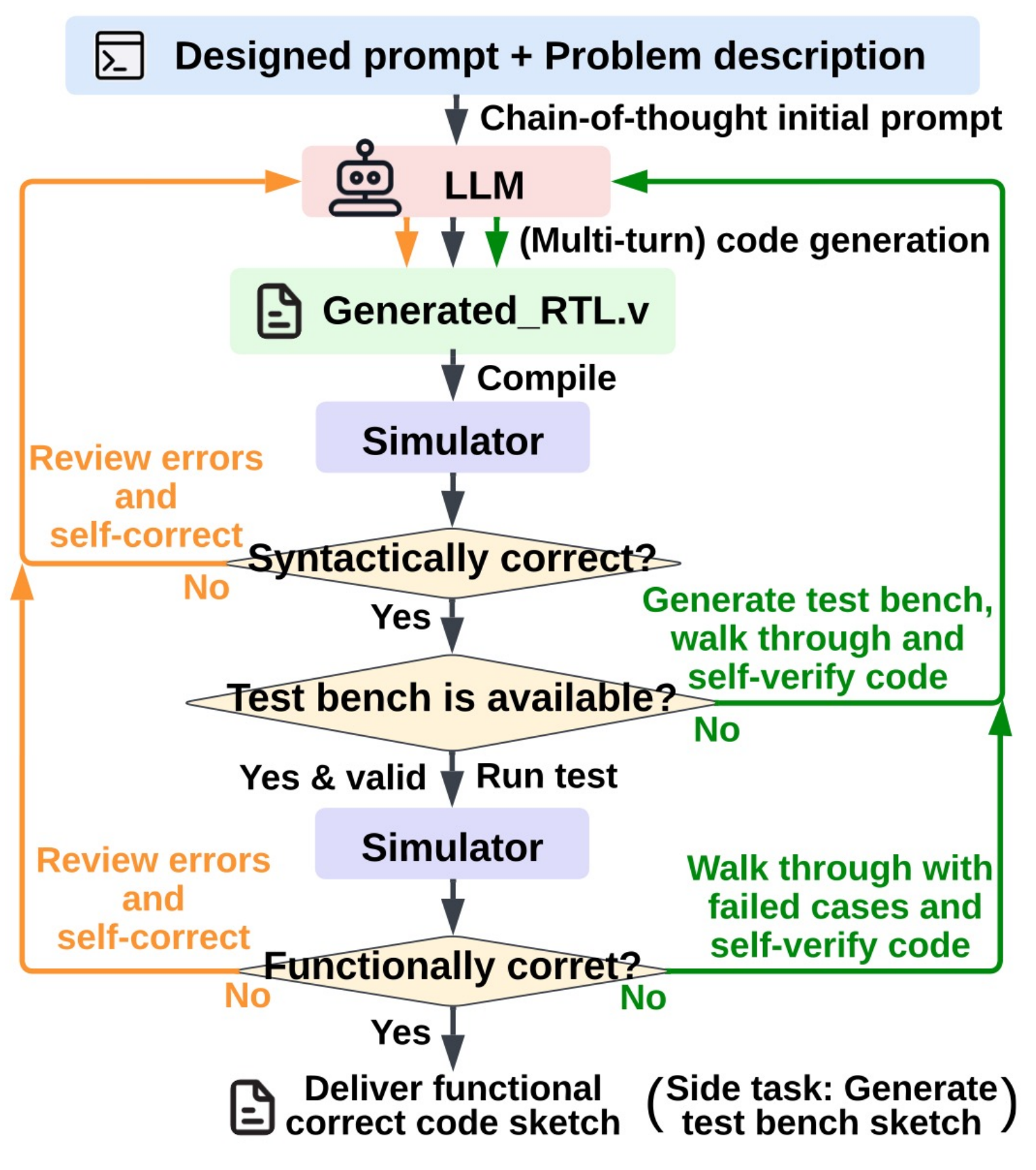

Designed prompt + Problem description
Chain-of-thought initial prompt
LLM
(Multi-turn) code generation
Generated_RTL.v
Compile
Simulator
Review errors and self-correct
Syntactically correct?
No
Yes
Generate test bench, walk through and self-verify code
Test bench is available?
No
Yes & valid
Run test
Simulator
Review errors and self-correct
Walk through with failed cases and self-verify code
Functionally corret?
No
No
Yes
Deliver functional correct code sketch
(Side task: Generate test bench sketch)

Table 1: Pass rate (%) comparison of RTL code generators on VerilogEval [16] and RTLLM [17] benchmarks.

| Model Type | Evaluated Model | VerilogEval-Machine | | | VerilogEval-Human | | | RTLLM[‡] pass@5 | |
|---|---|---|---|---|---|---|---|---|---|
| | | pass@1 | pass@5 | pass@10 | pass@1 | pass@5 | pass@10 | Syntax(%) | Func(%) |
| Open-Source Model | CodeGen2-16B [20] | 5.00 | 9.00 | 13.9 | 0.90 | 4.10 | 7.25 | 72.4 | 6.90 |
| | CodeGen-Verilog-16B [28] | 44.0 | 52.6 | 59.2 | 30.3 | 43.9 | 49.6 | 86.2 | 24.1 |
| Closed-Source Model | ChipNeMo-13B [15][†] | 43.4 | N/A | N/A | 22.4 | N/A | N/A | N/A | N/A |
| | ChipNeMo-70B [15][†] | 53.8 | N/A | N/A | 27.6 | N/A | N/A | N/A | N/A |
| | verilog-sft-16B [16][†] | 46.2 | 67.3 | 73.7 | 28.8 | 45.9 | 52.8 | N/A | N/A |
| | Claude-3 [4] | 55.3 | 63.8 | 69.4 | 34.4 | 48.3 | 53.4 | 93.1 | 55.2 |
| | GPT-3.5 | 46.7 | 69.1 | 74.1 | 26.7 | 45.8 | 51.7 | 89.7 | 37.9 |
| | GPT-4 | 60.0 | 70.6 | 73.5 | 43.5 | 55.8 | 58.9 | **100** | 65.5 |
| VeriAssist | Ours + Claude-3 | 63.8 | 70.4 | 78.4 | 41.6 | 55.5 | 62.5 | 96.6 | 65.5 |
| | Improvement (Δ)* | +8.5 | +6.6 | +9.0 | +7.2 | +7.2 | +9.1 | +3.5 | +10.3 |
| | Ours + GPT-3.5 | 55.3 | 76.5 | 80.1 | 34.4 | 51.3 | 58.9 | 93.1 | 48.3 |
| | Improvement (Δ)* | +8.6 | +7.4 | +6.0 | +7.7 | +5.5 | +7.2 | +3.4 | +10.4 |
| | Ours + GPT-4 | **67.5** | **78.3** | **83.2** | **50.5** | **62.8** | **69.2** | **100** | **75.9** |
| | Improvement (Δ)* | +7.5 | +7.7 | +9.7 | +7.0 | +7.0 | +10.3 | 0.0 | +10.4 |
| Ablation Study | Self-Verification + GPT-4 | 63.8 | 73.2 | 78.4 | 48.3 | 58.9 | 64.7 | 96.6 | 69.0 |
| | Self-Correction + GPT-4 | 62.5 | 72.2 | 77.2 | 47.1 | 58.9 | 66.0 | 100 | 69.0 |

**Takeaways:**

- VeriAssist **suggests** accurate code sketch, testbench with test cases
- VeriAssist **reduces** human intervention and improves productivity
- The proposed process of generating test benches, and self-code walk-throughs significantly improves the correctness of RTL code

**Evaluation:**

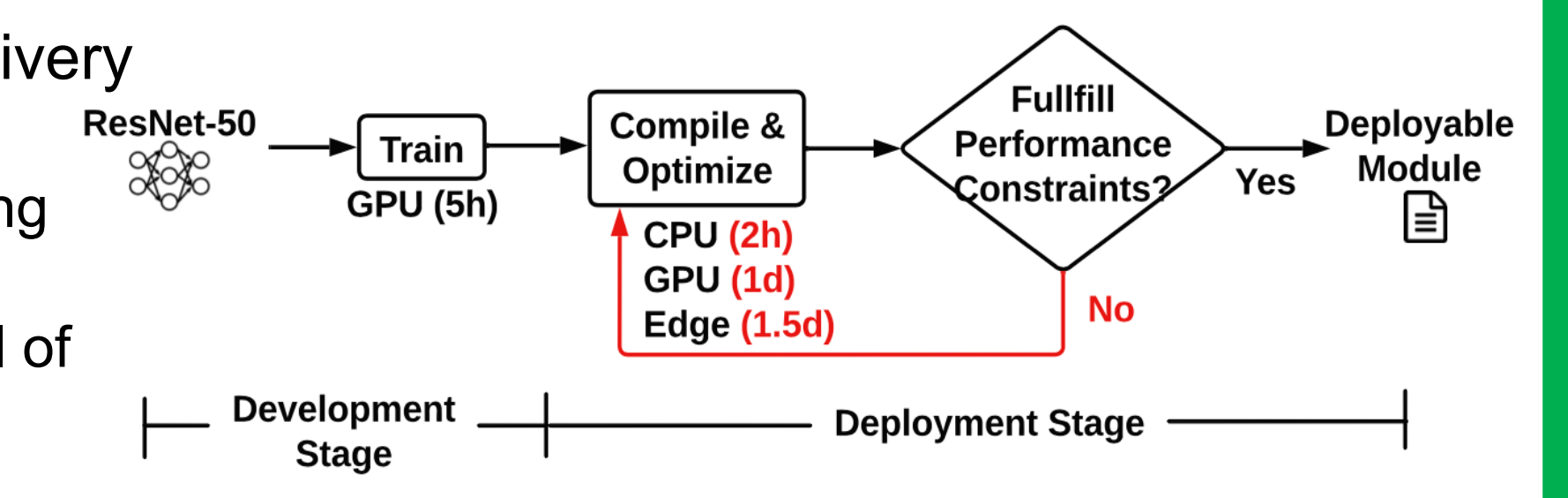
- Metrics: syntax pass rate, functionality pass rate, PPA
- pass@k: a problem is considered solved if any of the k samples pass the unit tests.
- VeriAssist suggests high-quality RTL code with an average pass@5 score of 72.3% and comparable PPA, along with corresponding test benches.

## Fasor: A Fast Tensor Program Optimization Framework for Efficient DNN Deployment

*Hanxian Huang[1], Xin Chen[2], Jishen Zhao[1]*
*University of California San Diego[1], Applied ML Group Intel Corp.[2]*

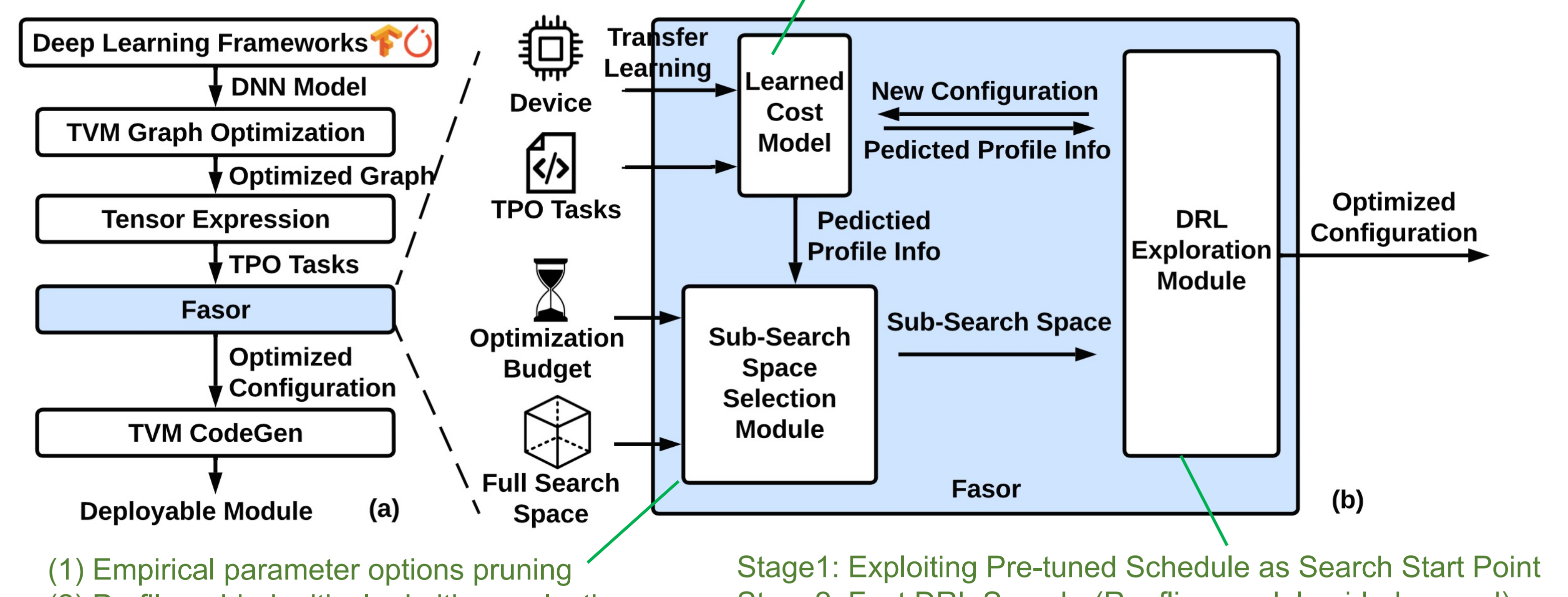
**Background and Motivation:**

- DNN deployment is becoming a bottleneck in DNN delivery
- Two inefficiencies in tensor program optimization:
  - Cost model training or transferring inefficiency, involving costly on-device measurement
  - Search sampling inefficiency, overlooking the potential of reusing pre-tuned schedules



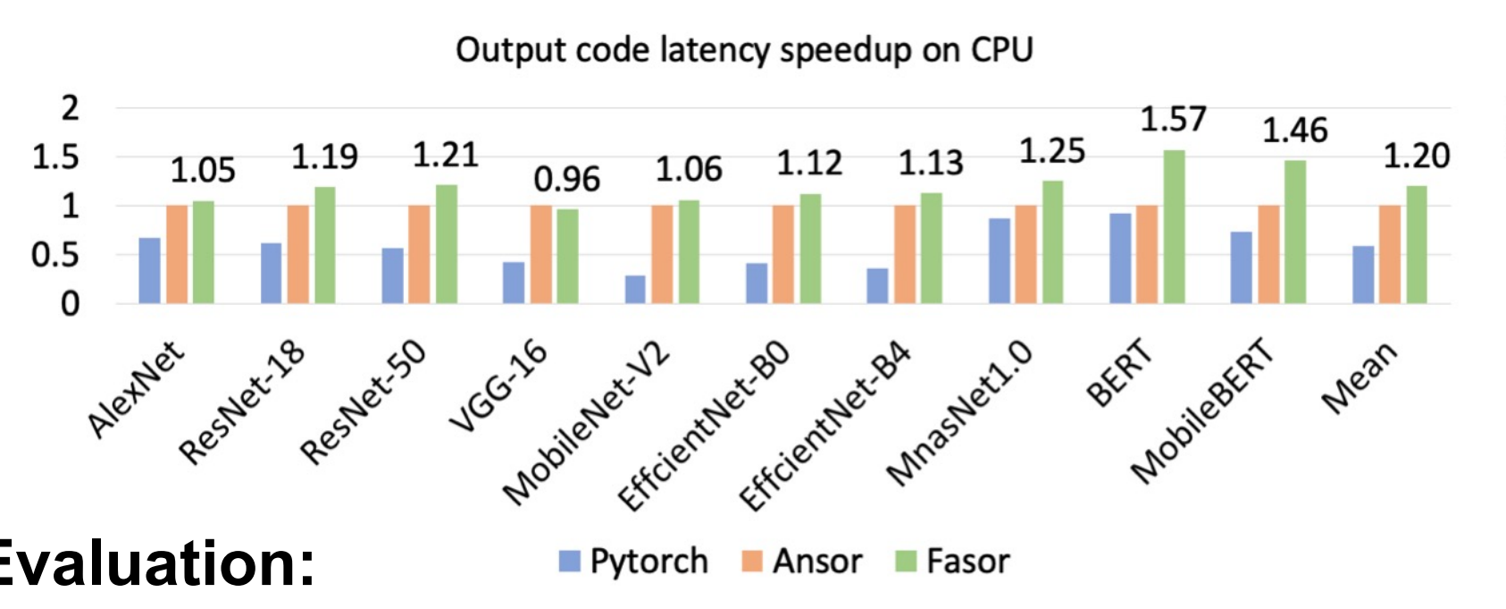
**Methodology:**

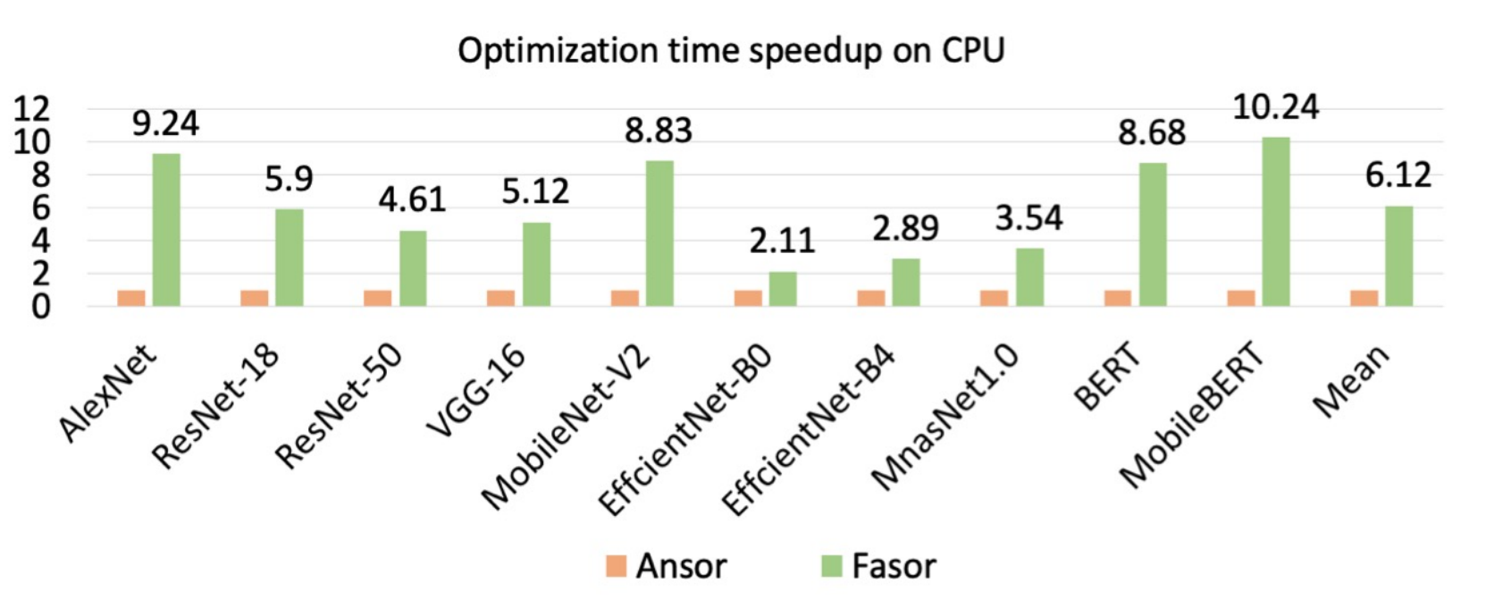
Hardware-transferrable cost model
(1) task feature: kernel/input/output shapes, serialized schedule configurations
(2) hardware feature: hardware specifications.
(3) a small calibration dataset with tasks that contribute distinct hardware-specific knowledge



(1) Empirical parameter options pruning
(2) Profile-guided critical primitives selection

Stage1: Exploiting Pre-tuned Schedule as Search Start Point
Stage2: Fast DRL Search. (Roofline model guided reward)





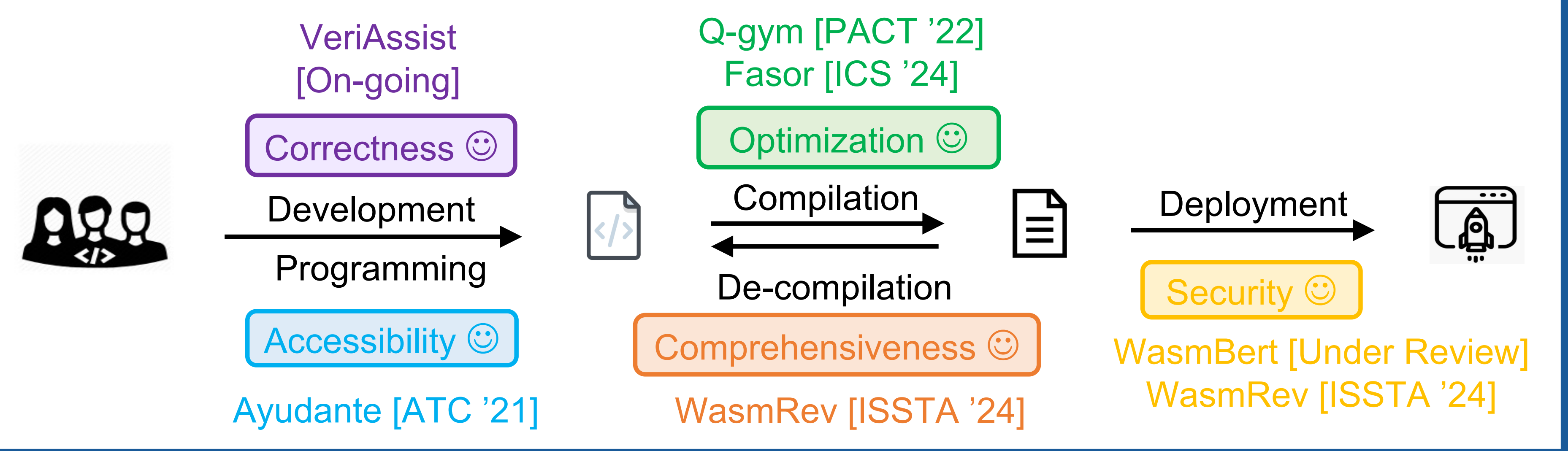
**Evaluation:**

- Fasor improves the compilation efficiency on the Intel CPU and NVIDIA GPU by up to 10.24× and 8.17×.
- Fasor delivers better or equal output code latency performance with 1.22× average speedup.
- Fasor effectively solves the cost model measurement (81%↓) and search (73%↓) inefficiencies.

**Takeaways:**

- Fasor provides a high-accurate hardware-transferable cost model that helps with configuration searching
- Fasor exploits tensor program similarity and introduces roofline model guidance achieve a faster and better configuration searching

## Overview



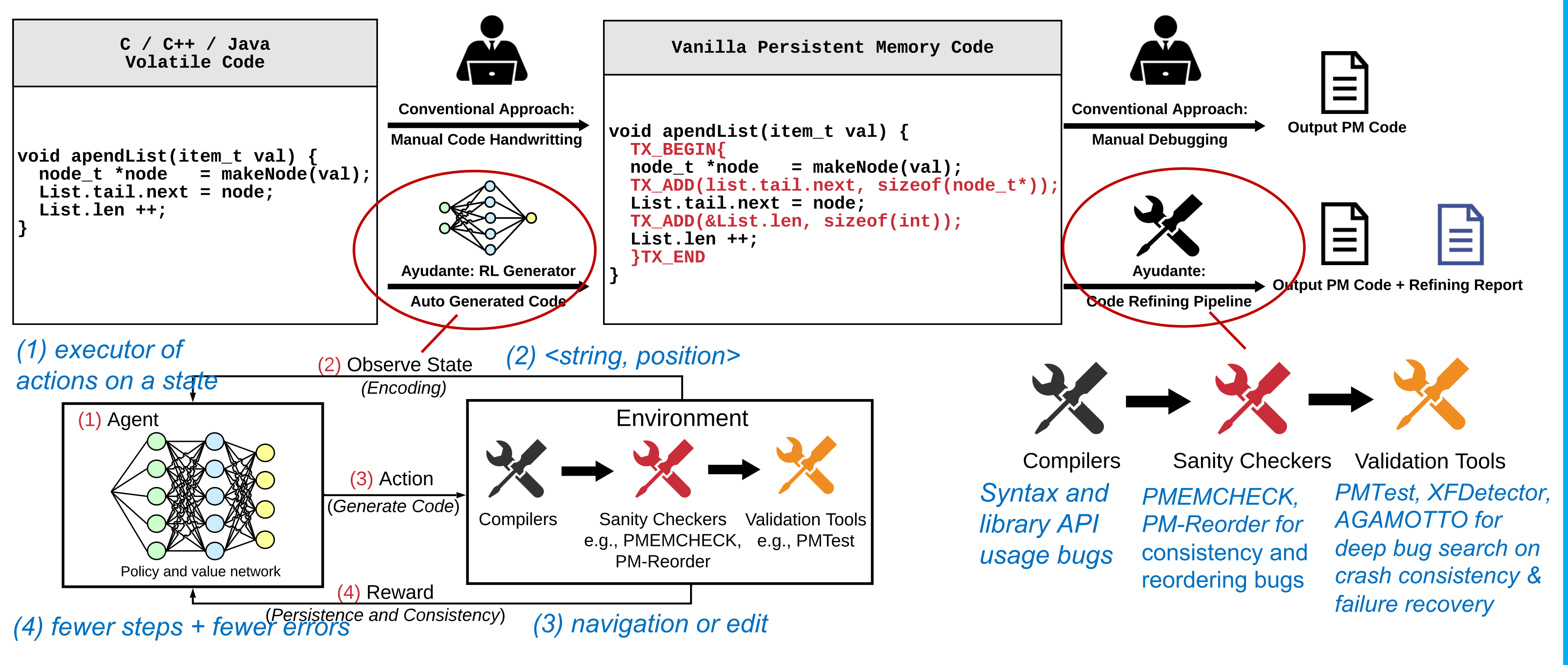
## Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming

Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson, Jishen Zhao
University of California San Diego

**Background and Motivation:**

- Persistent Memory (PM)
  - Comparable performance of DRAM + Persistence property of storage Persistent Memory (PM)
- PM-aware programming
  - Adopt PM library to maintain crash consistency and recover failure
- Challenges in PM-aware programming
  - Non-trivial labor effort, error-prone
  - Require detailed PM programming knowledge

**Methodology:**



```
C / C++ / Java
Volatile Code

void apendList(item_t val) {
  node_t *node    = makeNode(val);
  List.tail.next = node;
  List.len ++;
}
```

```
Vanilla Persistent Memory Code

void apendList(item_t val) {
  TX_BEGIN{
  node_t *node    = makeNode(val);
  TX_ADD(list.tail.next, sizeof(node_t*));
  List.tail.next = node;
  TX_ADD(&List.len, sizeof(int));
  List.len ++;
  }TX_END
}
```

Conventional Approach:
Manual Code Handwritting
Ayudante: RL Generator
Auto Generated Code
Conventional Approach:
Manual Debugging
Output PM Code
Ayudante:
Code Refining Pipeline
Output PM Code + Refining Report
(1) executor of actions on a state
(2) Observe State
(Encoding)
(2) <string, position>
(1) Agent
Policy and value network
(3) Action
(Generate Code)
Environment
Compilers
Sanity Checkers e.g., PMEMCHECK, PM-Reorder
Validation Tools e.g., PMTest
(4) Reward
(Persistence and Consistency)
(4) fewer steps + fewer errors
(3) navigation or edit
Compilers
Sanity Checkers
Validation Tools
Syntax and library API usage bugs
PMEMCHECK, PM-Reorder for consistency and reordering bugs
PMTest, XFDetector, AGAMOTTO for deep bug search on crash consistency & failure recovery

**An example of generated code:**

```
int64_t Queue::pop(){
  int64_t ret = 0;
  auto pool = pmem::obj::pool_by_vptr(this);
  obj::transaction::run(pool, [this, &ret] {
    if (head == nullptr)
      throw std::runtime_error("Empty queue");
    ret = head->value;
    auto n = head->next;
    obj::delete_persistent<Node>(head);
    head = n;
    if (head == nullptr) tail = nullptr;
  });
  return ret;
}
```

Navigation Action ❶ ❷ ❹ ❺ ❻ ❽ ❾ ⓫ ⓬ — Edit Action ❸ ❼ ❿

**Evaluation:**

- Accuracy/Correctness: PM checker pass rate (CPR)
- Labor effort reduction: #line of code (LOC)
- Performance compared to expert code (not shown here)

| Testing Set | Checkers in Environment | CPR | LOC |
|---|---|---|---|
| Microbenchmarks and KV store application | PMEMCHECK | 87.5% | 12.3% |
| | PMEMCHECK & PM-Reorder | **100%** | 13.4% |
| | PMEMCHECK & PM-Reorder & PMTest | **100%** | 13.8% |
| Leetcode solution | PMEMCHECK | 60.2% | 12.5% |
| | PMEMCHECK & PM-Reorder | 62.1% | 13.1% |
| | PMEMCHECK & PM-Reorder& PMTest | 78.7% | 13.4% |

**Takeaways:**

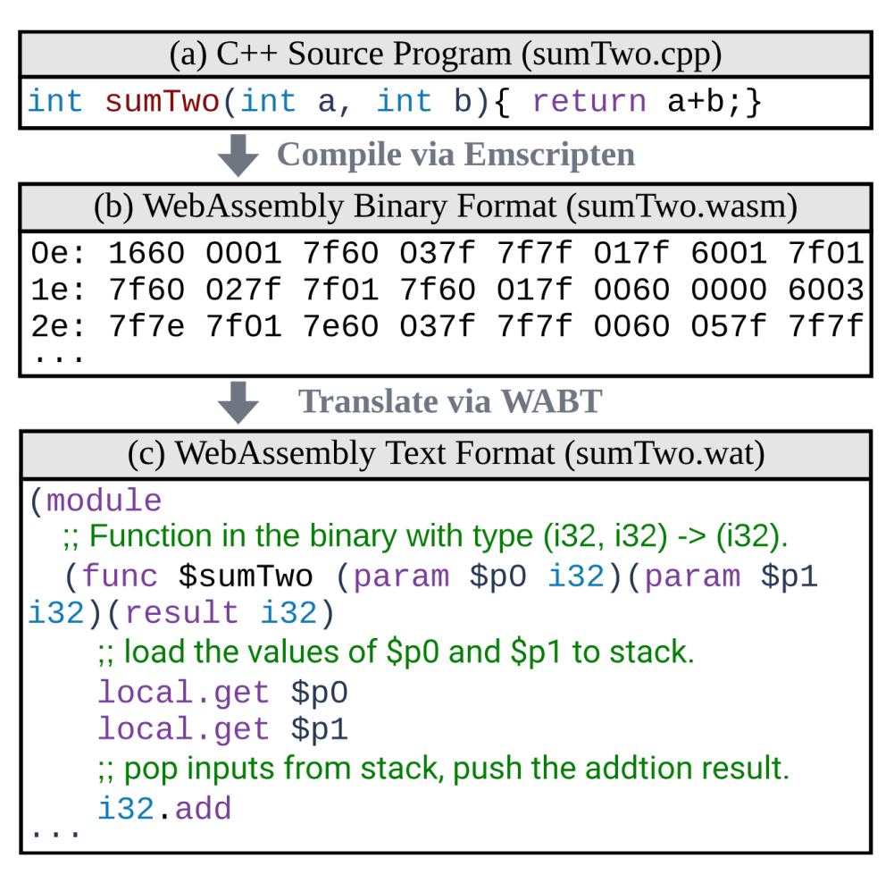
- Ayudante can **assist with** sophisticated PM-programming tasks through efficient PM code generation and code refining
- Ayudante **improves the accessibility of** domain-specific programming
- More insights: Monte Carlo tree-search (search efficiency); knowledge transferable among PLs); validation tools are critical

## Multi-modal Learning for WebAssembly Reverse Engineering

Hanxian Huang, Jishen Zhao
University of California San Diego

**Background and Motivation:**

- WebAssembly (Wasm)
  - A novel assembly-like bytecode format; compiled from source code in high-level languages (e.g., C / C++, Rust); stack machine architecture
- WebAssembly code comprehension is necessary
  - Many (malicious) Wasm modules are distributed through third-party services
  - Enabling debugging, checking vulnerabilities and maintenance
- Challenges in WebAssembly comprehension
  - Lacking high-level information, e.g., limited data types
  - Tracking stack behavior is cumbersome and error-prone



(a) C++ Source Program (sumTwo.cpp)

```
int sumTwo(int a, int b){ return a+b;}
```

Compile via Emscripten

(b) WebAssembly Binary Format (sumTwo.wasm)

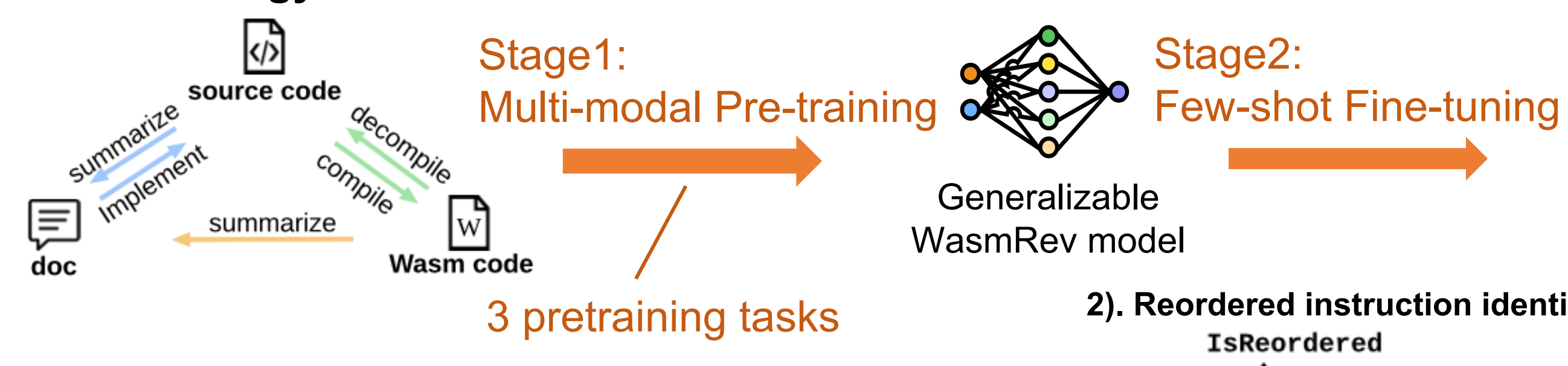
```
0e: 1660 0001 7f60 037f 7f7f 017f 6001 7f01
1e: 7f60 027f 7f01 7f60 017f 0060 0000 6003
2e: 7f7e 7f01 7e60 037f 7f7f 0060 057f 7f7f
...
```

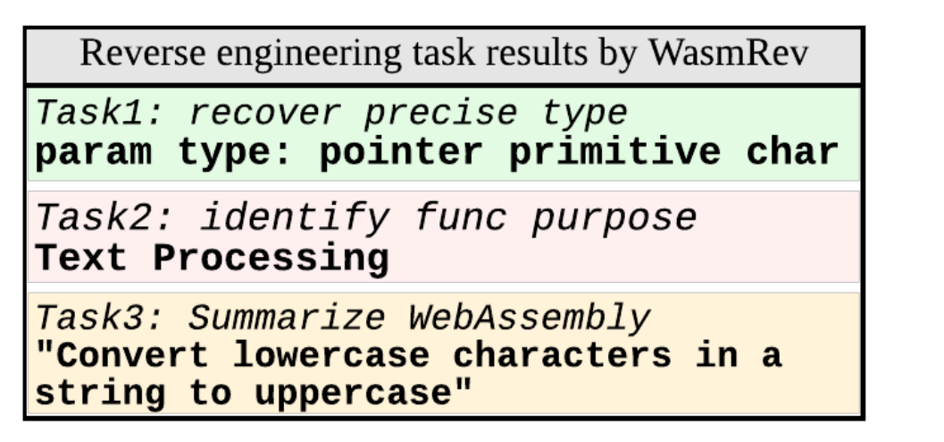
Translate via WABT

(c) WebAssembly Text Format (sumTwo.wat)

```
(module
  ;; Function in the binary with type (i32, i32) -> (i32).
  (func $sumTwo (param $p0 i32)(param $p1
i32)(result i32)
    ;; load the values of $p0 and $p1 to stack.
    local.get $p0
    local.get $p1
    ;; pop inputs from stack, push the addtion result.
    i32.add
...
```

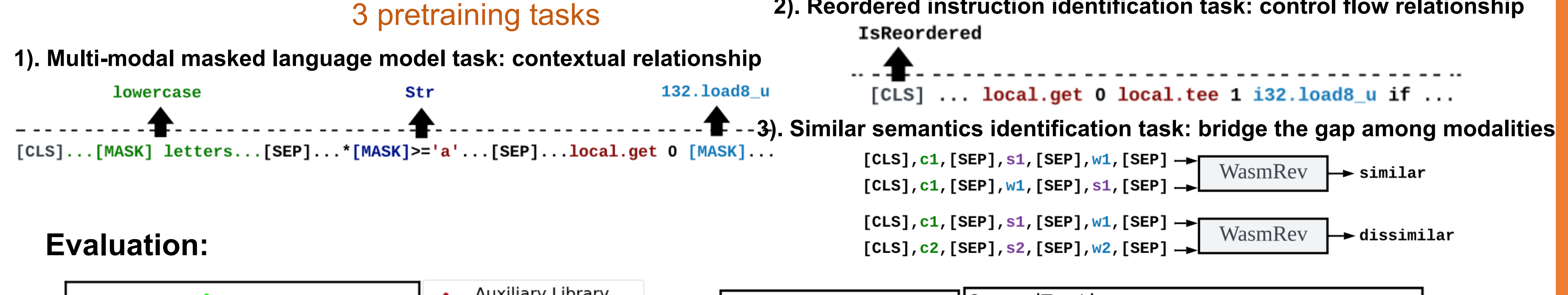
**Methodology:**



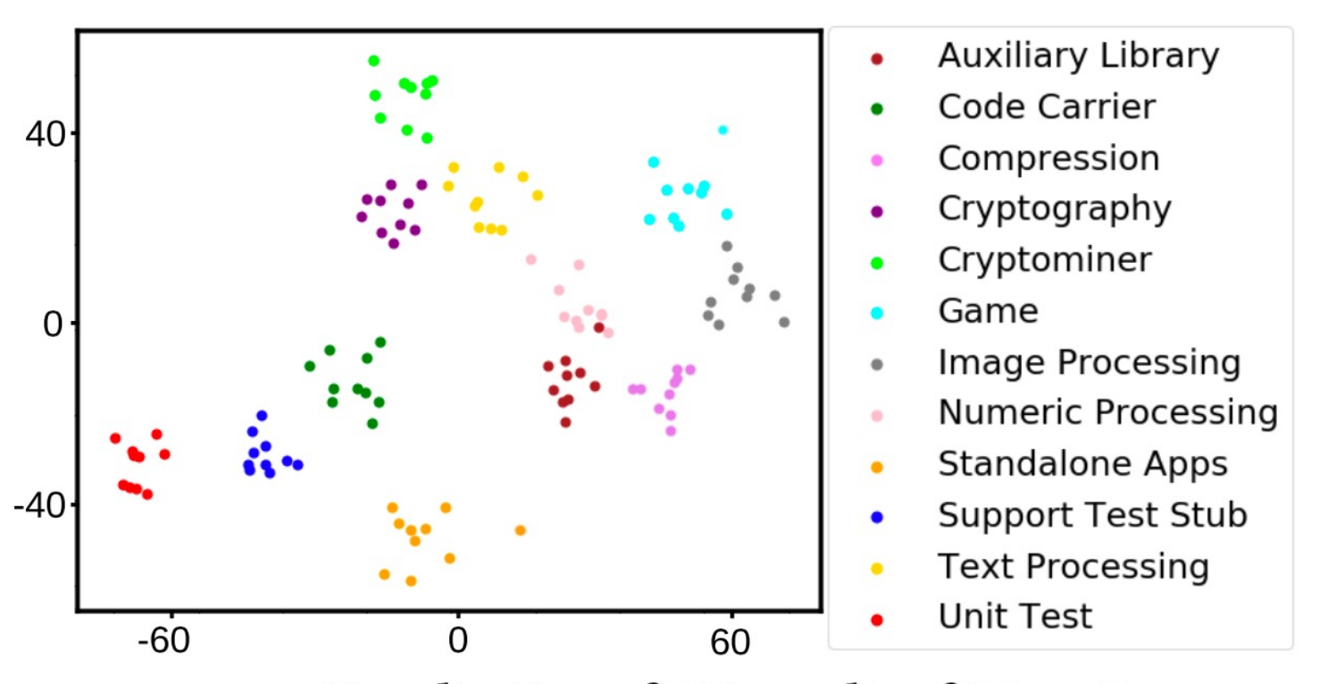


Reverse engineering task results by WasmRev

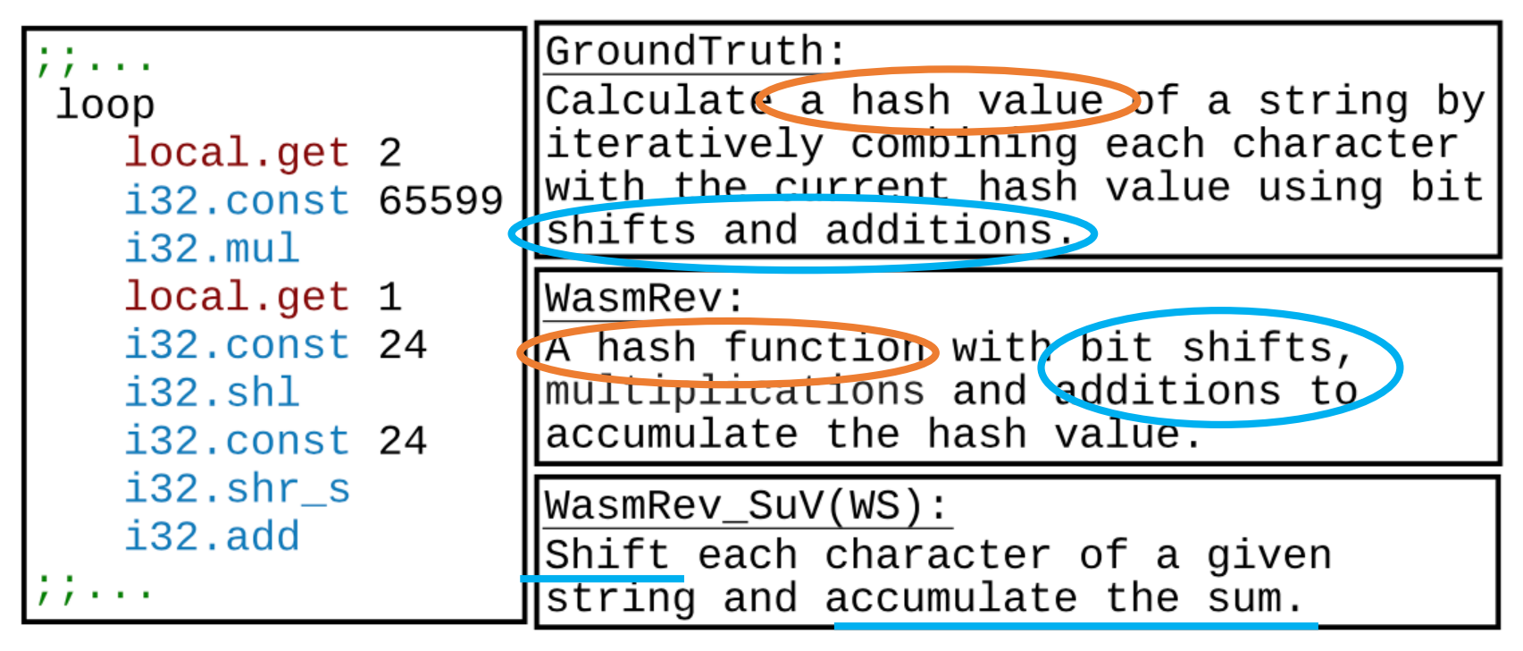
```
Task1: recover precise type
param type: pointer primitive char

Task2: identify func purpose
Text Processing

Task3: Summarize WebAssembly
"Convert lowercase characters in a
string to uppercase"
```


1). Multi-modal masked language model task: contextual relationship
lowercase
Str
132.load8_u
[CLS]...[MASK] letters...[SEP]...*[MASK]>='a'...[SEP]...local.get 0 [MASK]...
2). Reordered instruction identification task: control flow relationship
IsReordered
[CLS] ... local.get 0 local.tee 1 i32.load8_u if ...
3). Similar semantics identification task: bridge the gap among modalities
[CLS],c1,[SEP],s1,[SEP],w1,[SEP]
[CLS],c1,[SEP],w1,[SEP],s1,[SEP]
WasmRev
similar
[CLS],c1,[SEP],s1,[SEP],w1,[SEP]
[CLS],c2,[SEP],s2,[SEP],w2,[SEP]
WasmRev
dissimilar

**Evaluation:**



Visualization of FPI results of WasmRev.



```
;;...
 loop
   local.get 2
   i32.const 65599
   i32.mul
   local.get 1
   i32.const 24
   i32.shl
   i32.const 24
   i32.shr_s
   i32.add
;;...
```

```
GroundTruth:
Calculate a hash value of a string by
iteratively combining each character
with the current hash value using bit
shifts and additions.

WasmRev:
A hash function with bit shifts,
multiplications and additions to
accumulate the hash value.

WasmRev_SuV(WS):
Shift each character of a given
string and accumulate the sum.
```

A case study of WS.

Accuracy results on type recovery

| Task | | Parameter Type Prediction | | | | | Return Type Prediction | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Type Language | $\mathcal{L}_{SW}$ | $\mathcal{L}_{SW}$, All Names | $\mathcal{L}_{SW}$, Simplified | $\mathcal{L}_{Eklavya}$ | $\mathcal{L}_{SW}, t_{low}$ not given | $\mathcal{L}_{SW}$ | $\mathcal{L}_{SW}$, All Names | $\mathcal{L}_{SW}$, Simplified | $\mathcal{L}_{Eklavya}$ | $\mathcal{L}_{SW}, t_{low}$ not given |
| SnowWhite[†] | Top-1 Acc | 44.5% | 18.6% | 65.1% | 87.9% | 43.4% | 57.7% | 40.6% | 60.6% | 76.3% | 50.7% |
| | Top-5 Acc | 75.2% | 27.1% | 86.2% | 100.0% | 74.3% | 80.5% | 47.3% | 87.9% | 100% | 81.2% |
| | Type Prefix Score | 1.47 | 1.31 | 1.62 | 0.88 | 1.45 | 1.37 | 1.00 | 1.38 | 0.76 | 1.02 |
| WasmRev | Top-1 Acc | **63.7%** | 40.2% | 80.6% | 93.4% | 62.8% | **74.9%** | 52.4% | 79.7% | 89.2% | 73.5% |
| | Top-5 Acc | 88.3% | 50.0% | 95.2% | 100.0% | 87.4% | 93.8% | 63.9% | 95.1% | 100% | 93.2% |
| | Type Prefix Score | 1.78 | 1.55 | 1.89 | 0.93 | 1.76 | 1.60 | 1.30 | 1.58 | 0.89 | 1.31 |

[†] The results reported in the SnowWhite paper.

**Takeaways:**

- WasmRev **assists** WebAssembly comprehension by providing high-level semantics
- WasmRev **relieves** the burden of both WebAssembly users and tool developers
- WasmRev is **data-efficient** and **transferable** to new tasks

**Selected Publications:**

- "Muti-modal Learning for WebAssembly Reverse Engineering", **Hanxian Huang**, Jishen Zhao. *In the Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), 2024*
- "Fasor: A Fast Tensor Program Optimization Framework for Efficient DNN Deployment", **Hanxian Huang**, Xin Chen, Jishen Zhao. *In the Proceedings of the International Conference on Supercomputing (ICS)), 2024*
- "Q-gym: An Equality Saturation Framework for DNN Inference Exploiting Weight Repetition", Cheng Fu, **Hanxian Huang**, Bram Wasti, Chris Cummins, Riyadh Baghdadi, Kim Hazelwood, Yuandong Tian, Jishen Zhao, and Hugh Leather. *In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 2022*
- "Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming", **Hanxian Huang**, Zixuan Wang, Juno Kim, Steven Swanson, and Jishen Zhao. *In the Proceedings of USENIX Annual Technical Conference (USENIX ATC), 2021*
- "Towards LLM-Powered Verilog RTL Assistant: Self-Correction and Self-Verification", **Hanxian Huang**, Zhenghan Lin, Zixuan Wang, Xin Chen, Ke Ding, Jishen Zhao (Under Review)
- "Neural WebAssembly Comprehension: A Transferable WebAssembly Learning Framework for Generalized Analysis Tasks", **Hanxian Huang**, Jishen Zhao (Under Review)