Carnegie Mellon University

Leveraging Wasm Instrumentation

Arjun Ramesh & Tianshu Huang

arjunr2@andrew.cmu.edu, tianshu2@andrew.cmu.edu

BOSCH

WebAssembly Research Center

Why Wasm as an instrumentation target?

Why Wasm as an instrumentation target?

	Source Code	LLVM IR	Binary	Wasm
Simple Representation	No	Mostly	No	Yes
Cross-platform	Yes	Almost[1]	No	Yes
Compiler-agnostic	Yes	LLVM[2]	No	Yes
Language-agnostic	No	Mostly	Yes	Yes
Doesn’t require source code	No	Yes	Yes	Yes
Easy to bringup	Yes	Yes	Yes	Eh

[1]LLVM IR carries some platform-specific information such as native integer sizes
[2]you can use any compiler as long as it’s LLVM

Wasm really is cross platform!

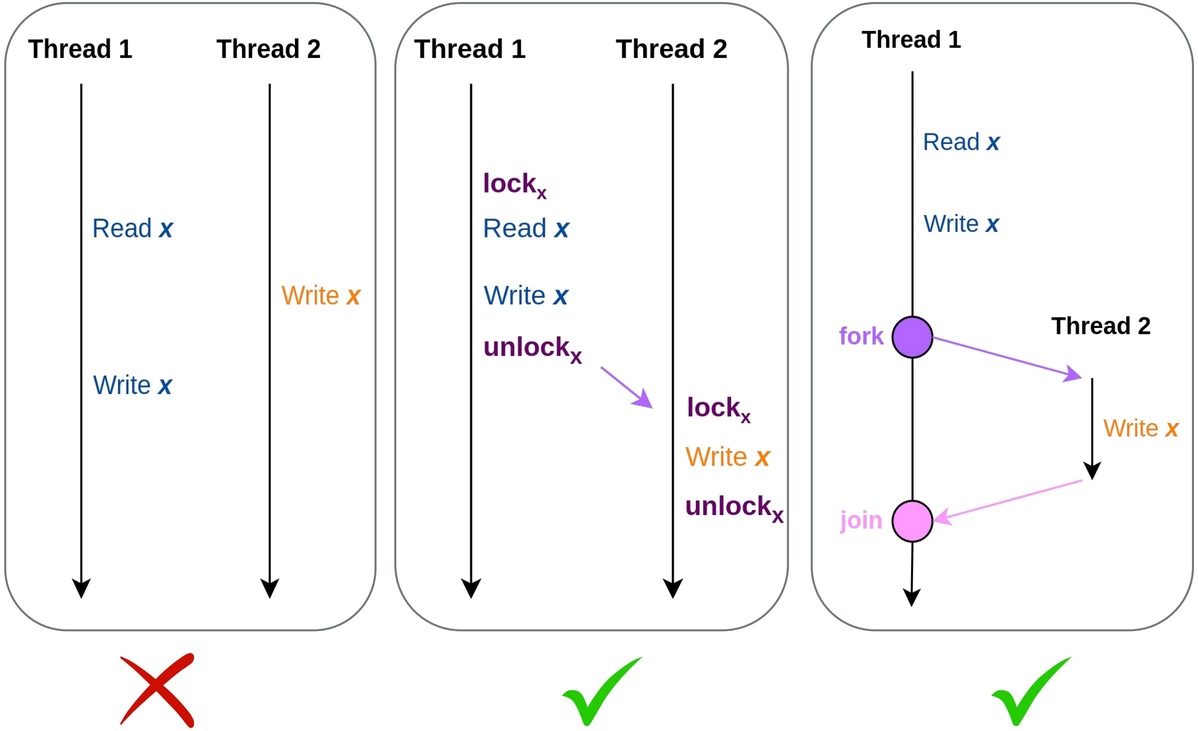
Contact us if you want access!

Wasm instrumentation for **Data Race Debugging**

Arjun Ramesh

Background: **Data-Race Conditions**

- **Shared** resource
- **Concurrent access** from multiple execution contexts
- At least one **write**
- **Inappropriate synchronization**



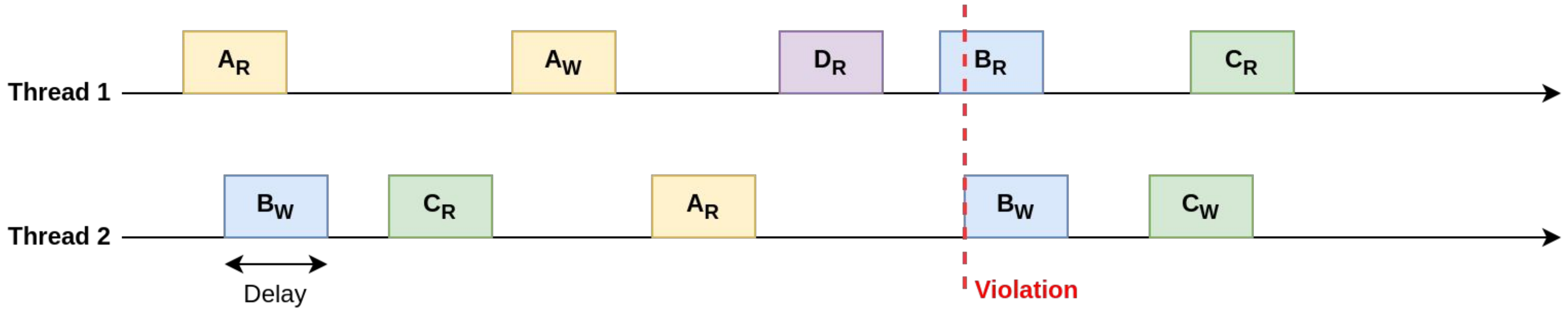
Background: **Data-Race Conditions**

Background: **Data-Race Detection Algorithms**

- **Static Detection:** Source code or bytecode analysis
- **Dynamic Detection:**
  - **Lockset Analysis:** Adhere to locking discipline
  - **Happens-before analysis:** Monitor sync primitives
  - **Trapped-delay injection:** Catch data-races red-handed

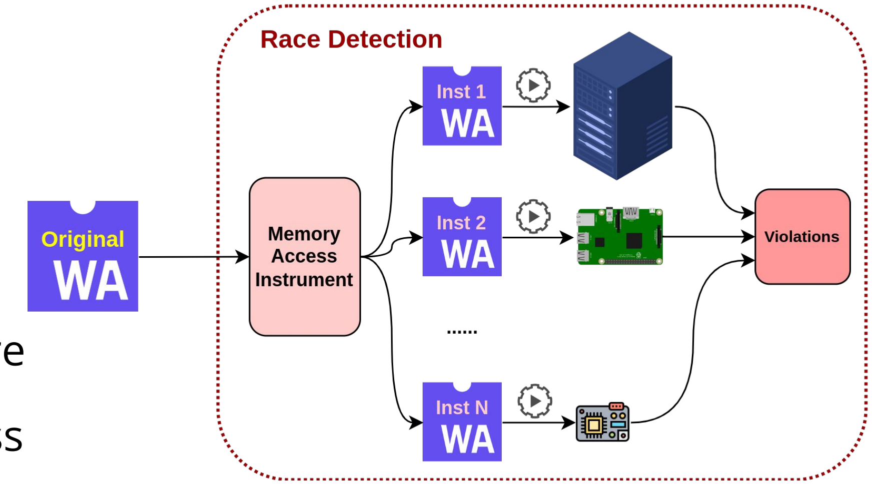
Inflexible for deployment

Lack of platform coverage

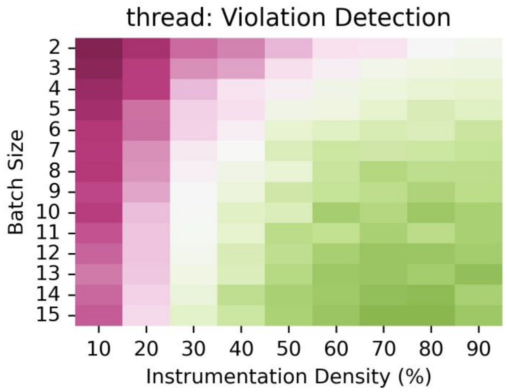


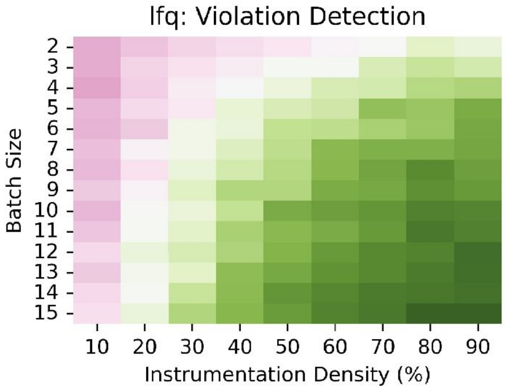
Solution: **Wasm for Data-Race Detection**

- **Language-agnostic**
- **Heterogeneous** bug-finding
- **Distributed, scalable** debugging infrastructure
- **Ease of adoption** across domains (e.g. real-time)

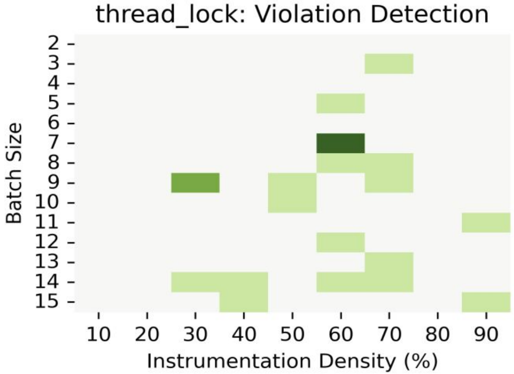


Why vary instrumentation density?

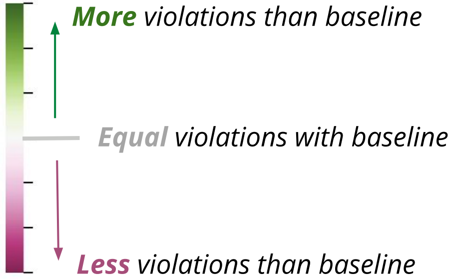




but...



Baseline: 1 run of 100% instrumentation density



Homogenous Testbed:
Intel NUC 11, core i7

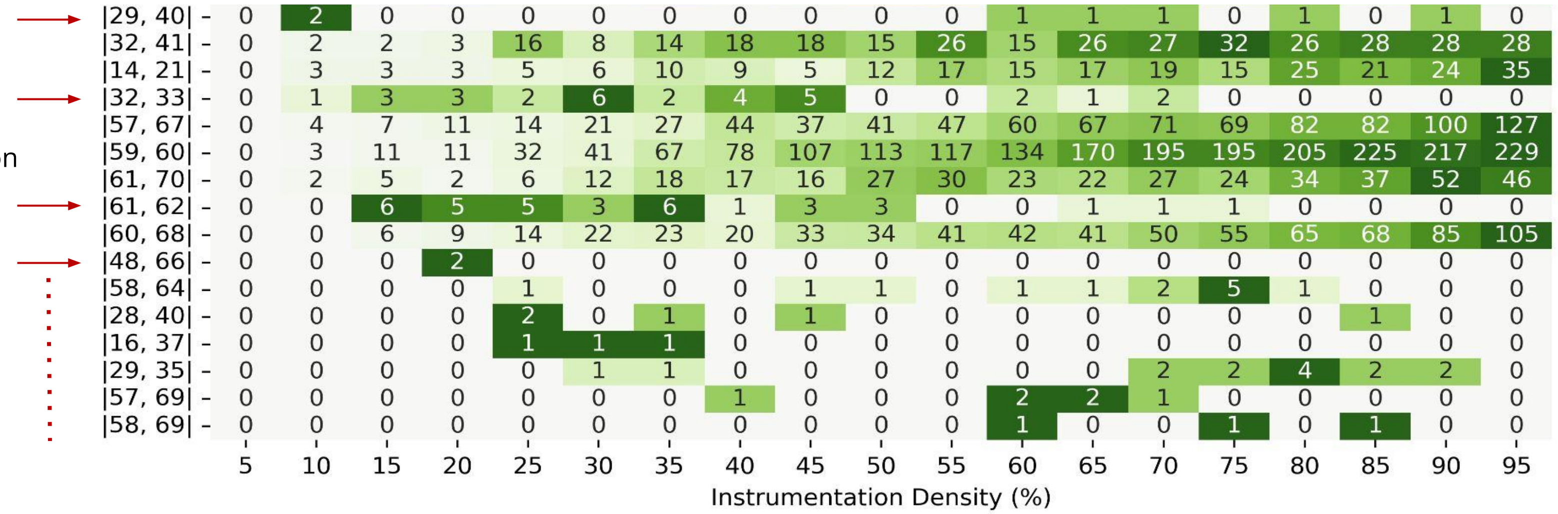
Tradeoff overhead for scalability

Rare bugs require very specific conditions

**500 runs performed for each instrumentation density (homogeneous)*

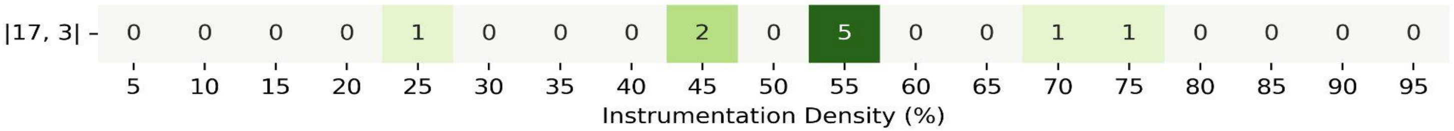
lfq

Unique Violation Pairs

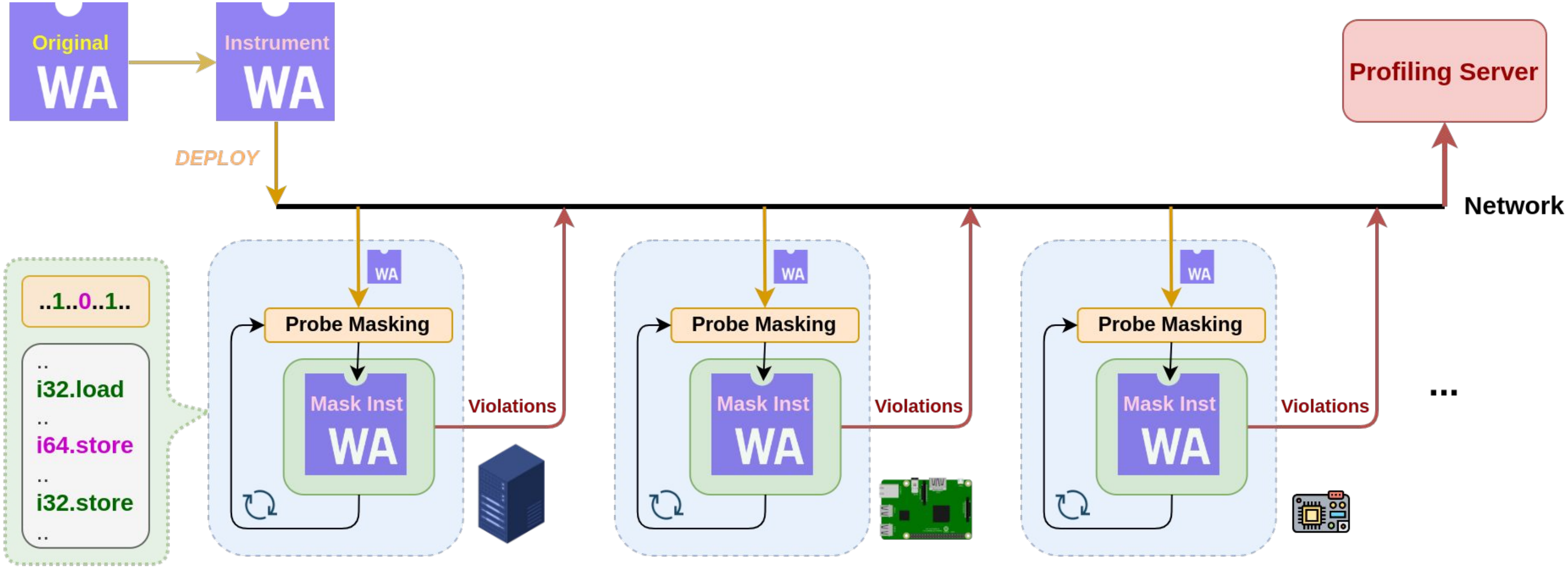


thread_lock

Unique Violation Pairs



Heterogeneous dynamic analysis infrastructure

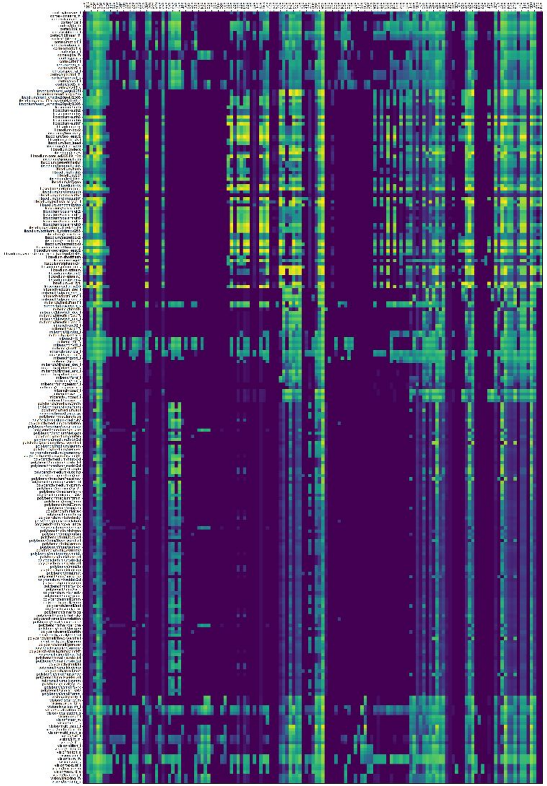


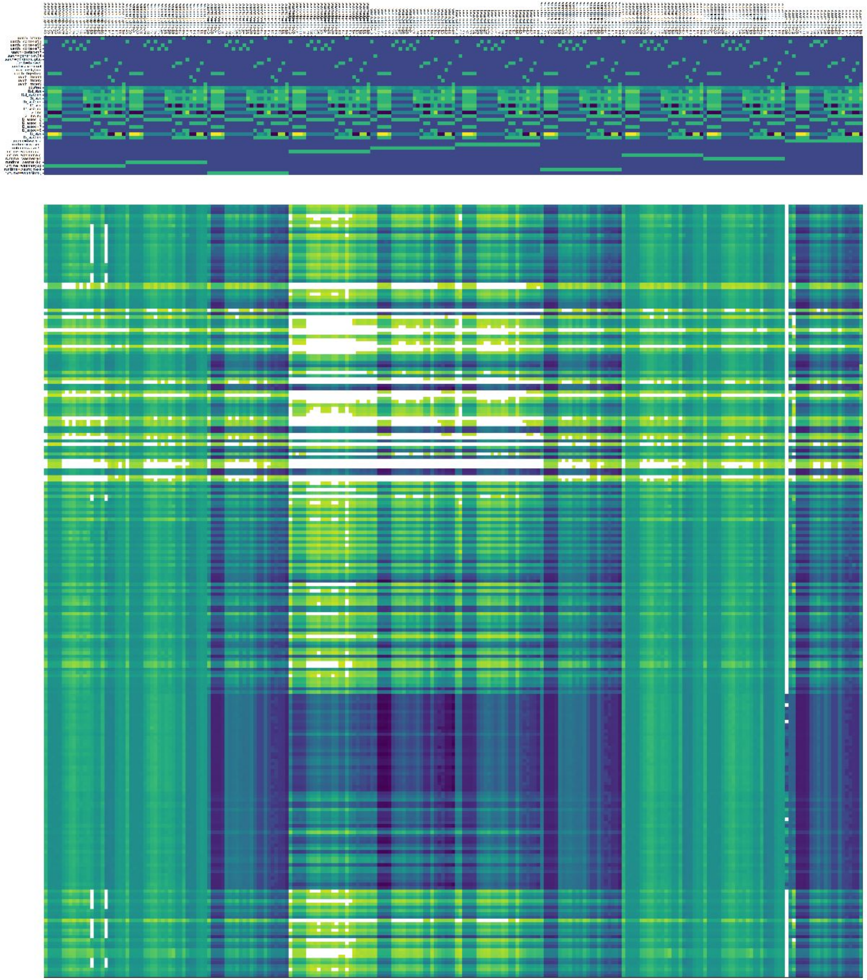
Platform Fuzzing = More Bugs? → **Low overhead, intelligent fuzzing**

Intermission: **WebAssembly Performance Analysis**

Tianshu Huang

Dataset

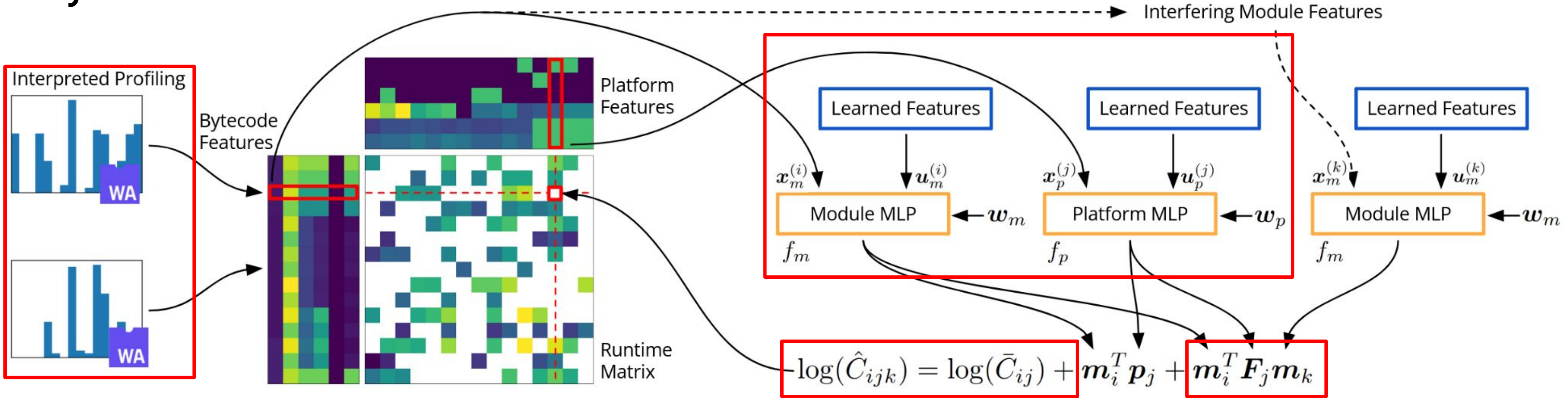




Method: **Matrix Factorization** (with Side Information)

3. "Two Tower" Model

1. Bytecode Features



2. Log-Residual Objective

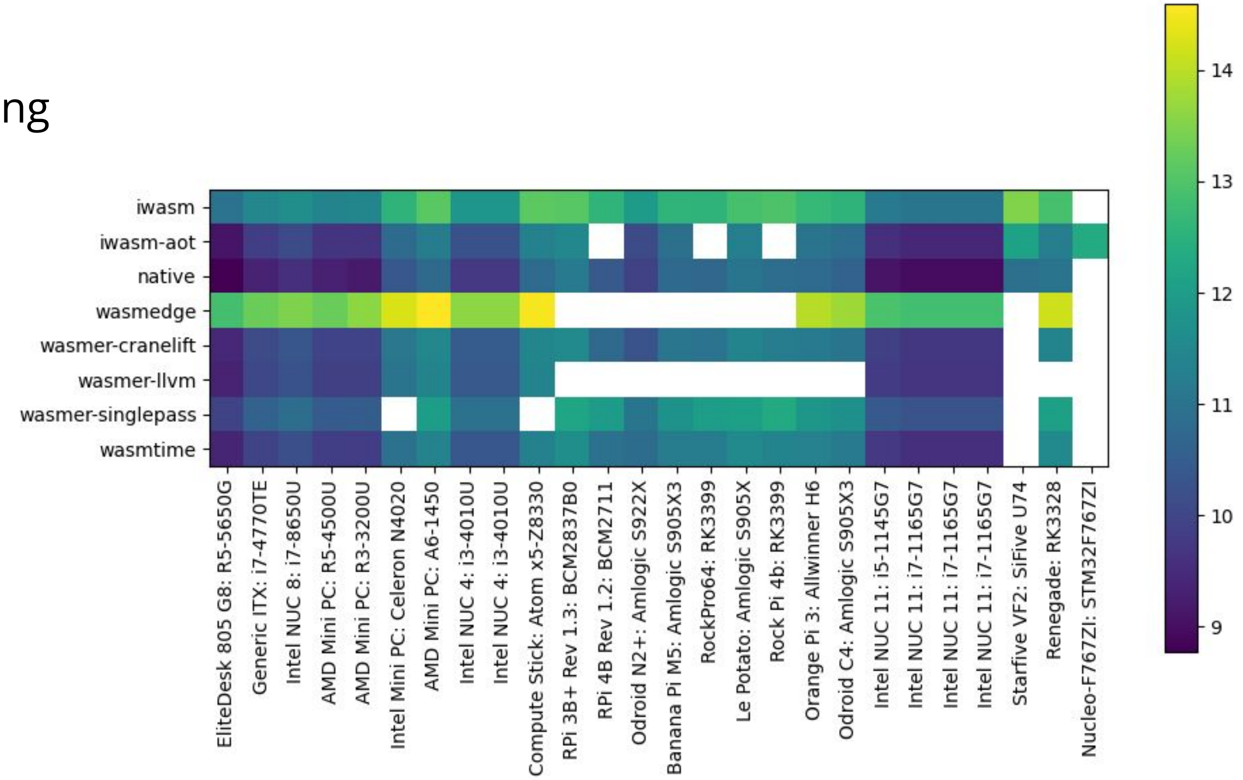
4. Interference Term

Log-Residual Objective

- Geometric Averaging

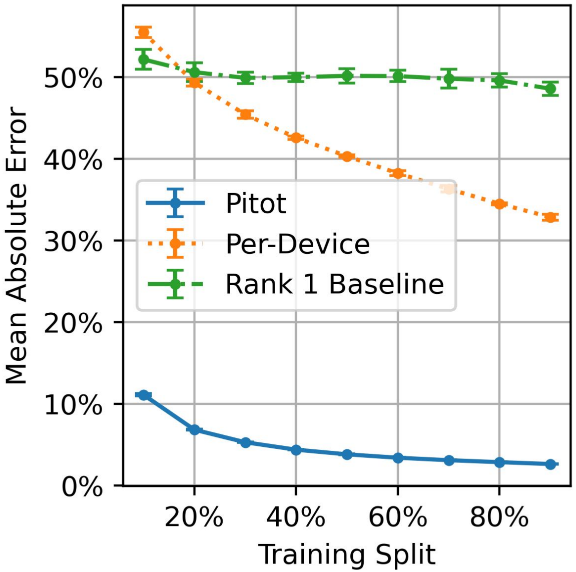
  = Log Arithmetic Averaging
- Residual Objective

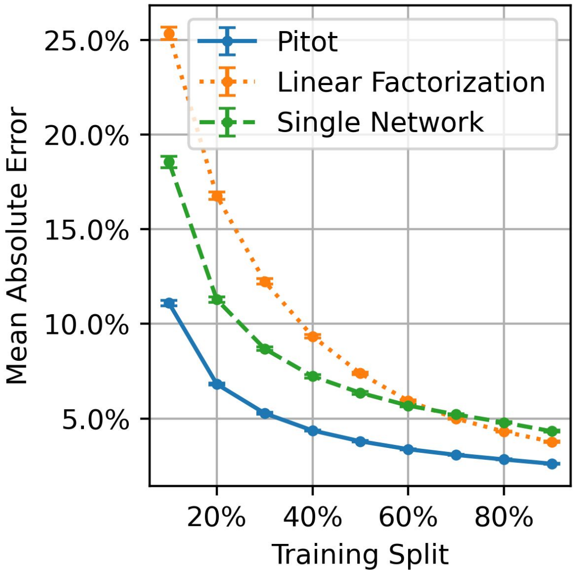
  = Normalize for scalar “speed” / “difficulty” first



It works

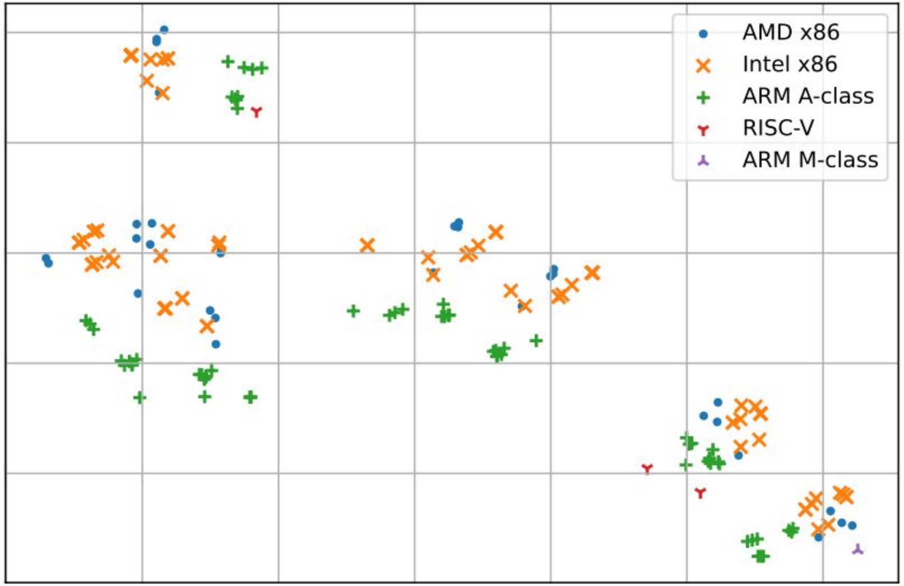
Pitot vs Baselines:



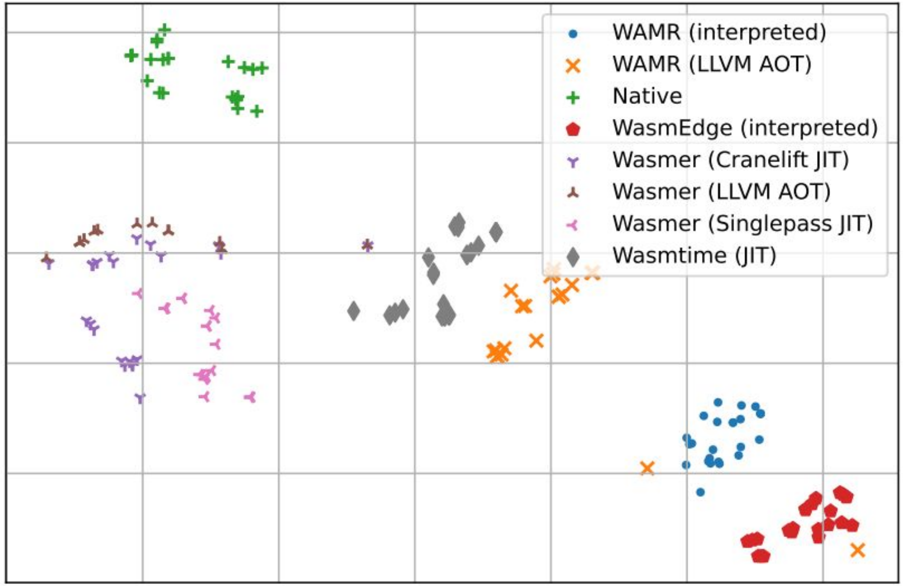


(2.6%)

Characterizing Platforms*



(a) Platforms by Microarchitecture Type



(b) Platforms by WebAssembly Runtime

*TSNE projection into 2 dimensions of the learned embeddings

Carnegie Mellon University

Wasm instrumentation for **Runtime Analysis**

Tianshu Huang

The Big Problem: **Data Dependence**

Motivation: **Code Coverage Instrumentation**

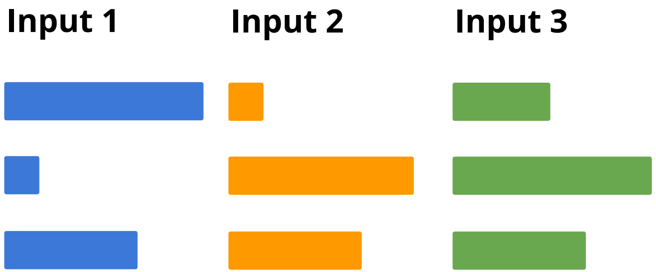
The problem:

- compute can **vary greatly with inputs**
- **can't understand** the input data

Our solution:

- **code frequency = input data** in all the ways that matter

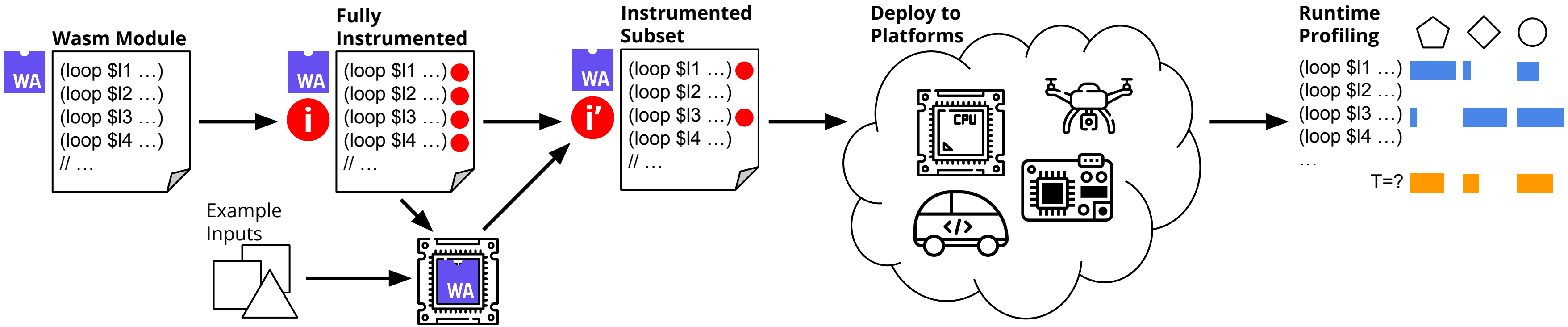
```
func(a, b) {
    for(i = 1..a)
        // computation A
    for(i = 1..f(b))
        // computation B

    // computation C
    return something;
}
```



Our Approach: **Code Frequency via Loop Counts**

- Instrument loops
- Remove highly correlated loops

- More looping = more overhead
  ⇒ remove in decreasing order of # loops

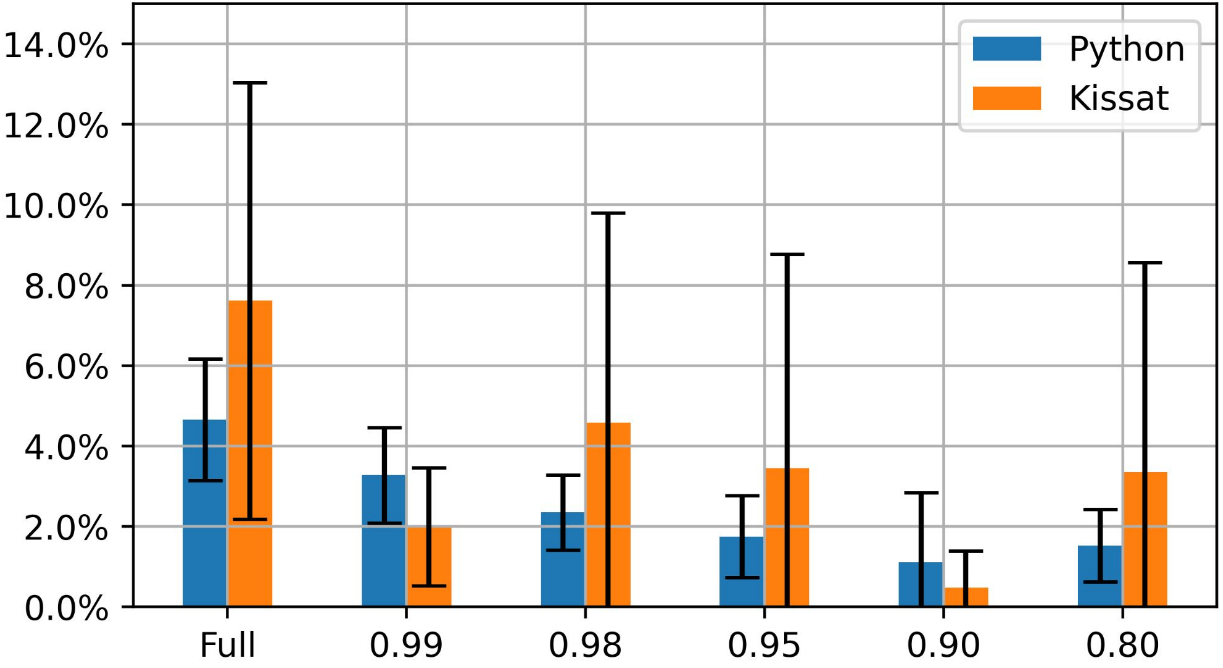


Results: **Overhead** (some rough numbers…)

Opcode counting overhead:

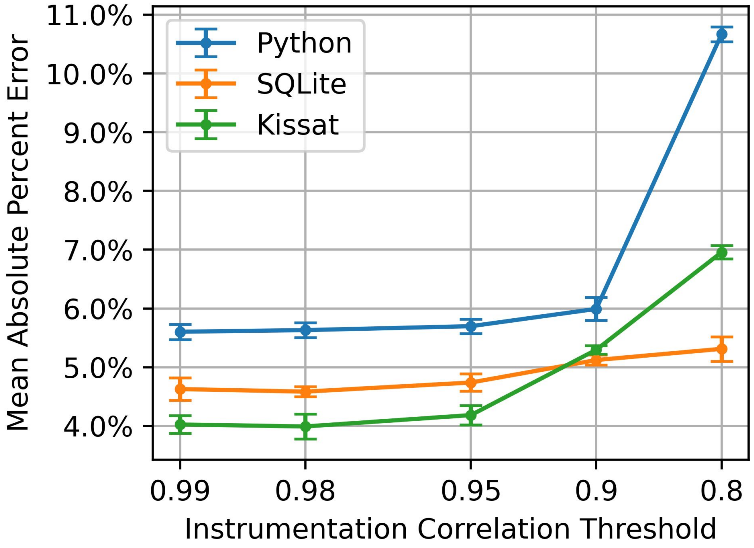
Python: 83%

Kissat: 174%

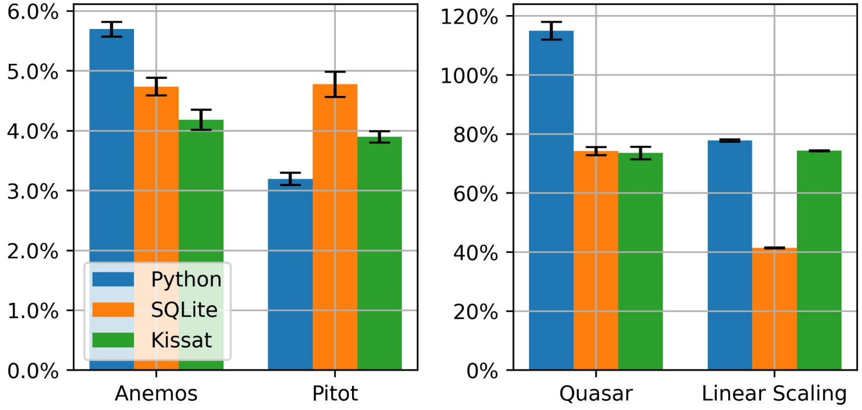


Results: **Prediction Accuracy**

- Can remove instrumentation with correlation >0.95



- Better than black box baselines
- Almost as good as a full opcode count



Conclusion

Correspondence to:

Arjun Ramesh

<arjunr2@andrew.cmu.edu>

Tianshu Huang

<tianshu2@andrew.cmu.edu>