Airlift

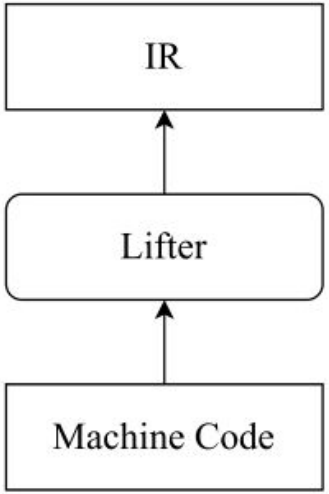
A Binary Lifter Based on a Machine-Readable Architecture Specification

Anders Choi
Advisor: Martin Fink
Chair of Computer Systems
https://dse.in.tum.de/

29.11.2024 – 30.05.2025

Motivation: Binary Lifting

- Critical component in a wide range of tasks
- Lacks reuse for custom IRs
- Highly complex and error-prone
- Error-sensitive



Binary lifter implementation is **critical** and **non-trivial**!

State-of-the-Art: Automated Lifter Generation

Cross-Architecture Lifter Synthesis leverages an existing lifter for one ISA to **inductively generate** a lifter for another ISA

SEFM 2018

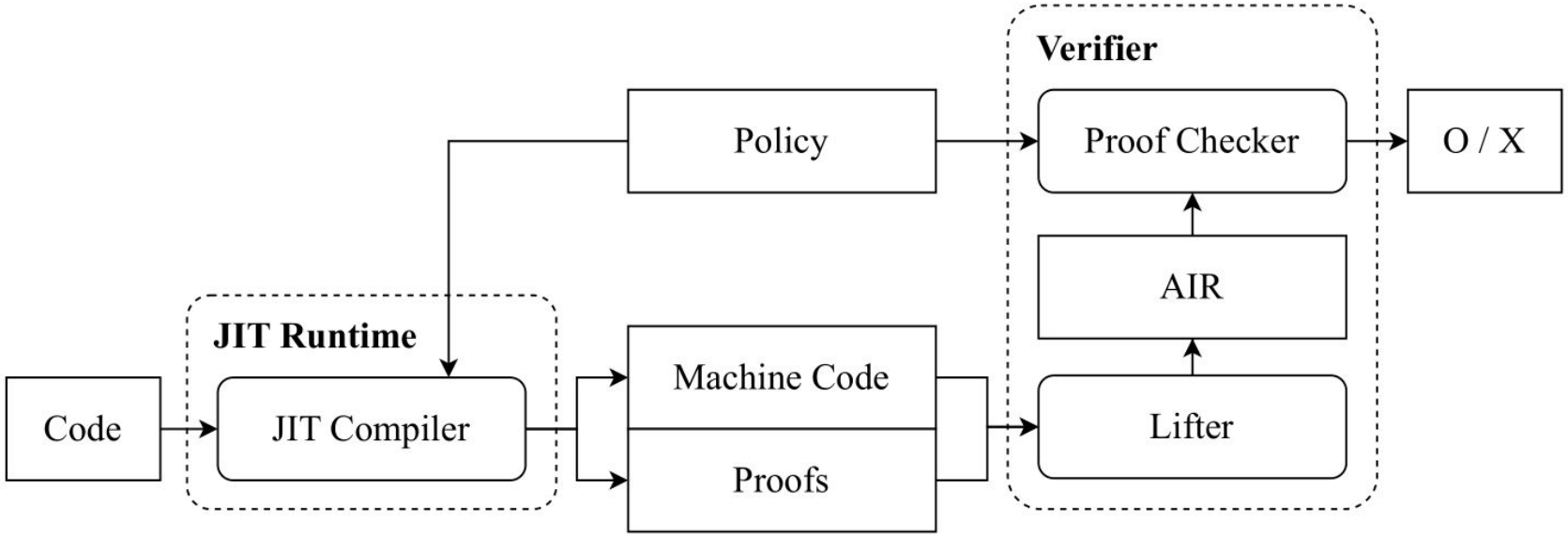
Forklift is a **neural lifter** that translates binary code to the target IR using a Transformer

COLM

Lift-Offline generates a lifter **from a formal specification**

SAS'24

Context: TrustNoJit (TNJ)



Context: AIR

A custom, lightweight IR for TNJ

- 3 data types
  - Unbounded integer
  - Fixed size integer
  - Boolean
- 37 instructions (vs. 100+ in LLVM IR)
- Block-based control flow

```
entry:
    v0 = i64.read_reg "x1"
    v1 = i64.read_reg "x2"
    v2 = i1.read_reg "n"
    v3 = bool.icmp.i1.eq v2, 0x1
    jumpif v3, block_0, block_1
block_0:
    jump block_2(v0)
block_1:
    jump block_2(v1)
block_2(v4: i64):
    write_reg.i64 v4, "x0"
```

Problem Statement

How can we generate a **precise** binary lifter for a **security-critical** system?

Challenges:

- Large volume of code
- Error-prone development
- Security-critical context

Airlift: Overview

Transpile a **machine-readable architecture specification** into a binary lifter

System design goals:

- High instruction coverage
- Semantic correctness
- ~~Performance and optimization~~

ASL: Arm's Machine-Readable Architecture Specification

```
__decode A64
case (29 +: 3, 24 +: 5, 0 +: 24) of
    ...
    when (_, 'x101x', _) =>
        case (
            31 +: 1, 30 +: 1, 29 +: 1, 28 +: 1, 25 +: 3,
            21 +: 4, 16 +: 5, 10 +: 6, 0 +: 10
        ) of
            ...
            when (_, _, _, '1', _, '0100', _, _, _) => // csel
                __field sf 31 +: 1
                __field op 30 +: 1
                __field S 29 +: 1
                __field Rm 16 +: 5
                __field cond 12 +: 4
                __field op2 10 +: 2
                __field Rn 5 +: 5
                __field Rd 0 +: 5
                case (sf, op, S, op2) of
                    when ('0', '0', '0', '00')
                        => __encoding aarch64_integer_conditional_select
                    ...
```

Instruction decode logic

```
__instruction aarch64_integer_conditional_select
    __encoding aarch64_integer_conditional_select
        __instruction_set A64
        __field sf 31 +: 1
        __field op 30 +: 1
        __field Rm 16 +: 5
        __field cond 12 +: 4
        __field o2 10 +: 1
        __field Rn 5 +: 5
        __field Rd 0 +: 5
        __opcode 'xx011010␣100xxxxx␣xxxx0xxx␣xxxxxxxx'
        __guard TRUE
        __decode
            integer d = UInt(Rd);
            integer n = UInt(Rn);
            integer m = UInt(Rm);
            integer datasize = if sf == '1' then 64 else 32;
            bits(4) condition = cond;
            boolean else_inv = (op == '1');
            boolean else_inc = (o2 == '1');
```

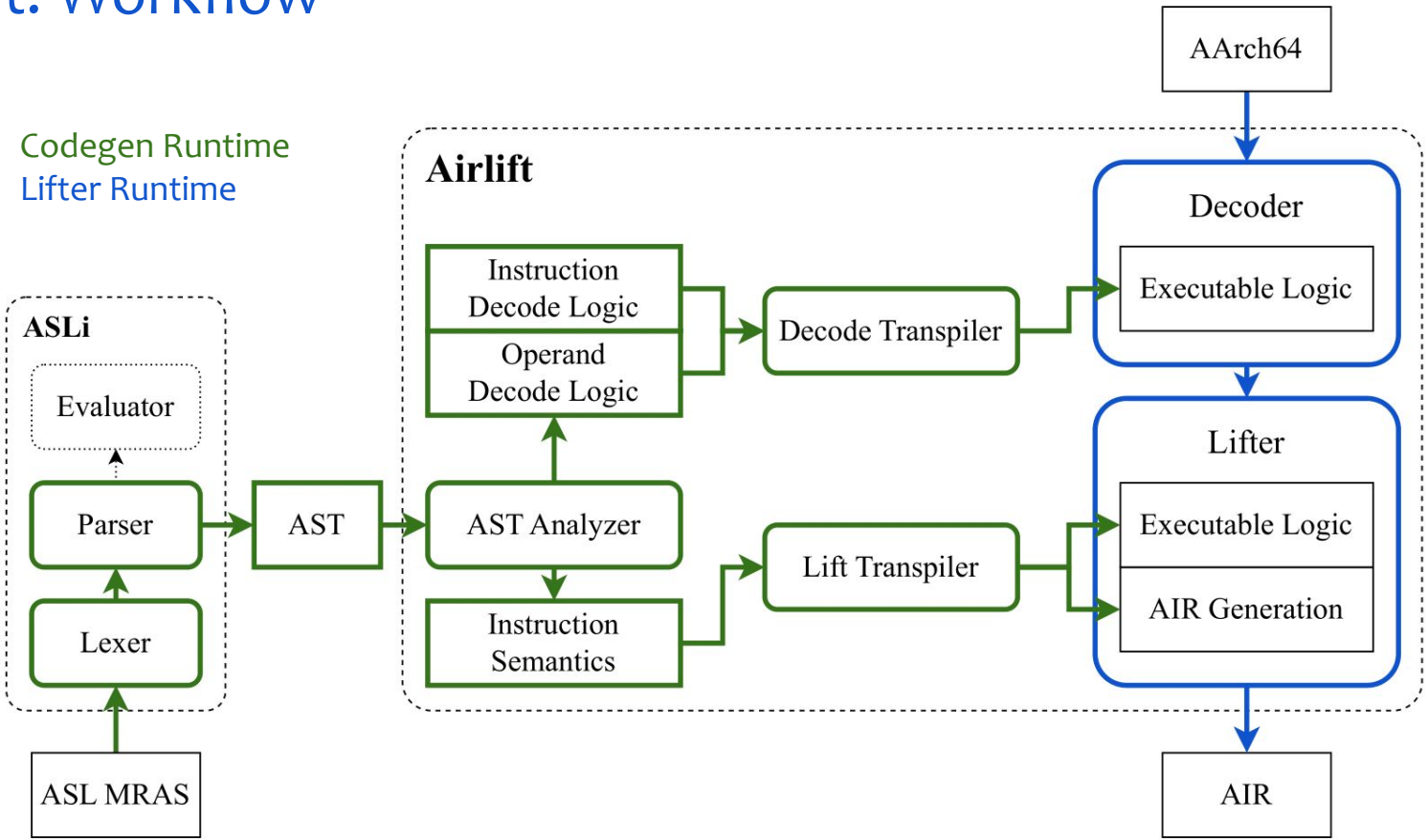
Operand decode logic

```
__execute
    bits(datasize) result;
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = X[m];

    if ConditionHolds(condition) then
        result = operand1;
    else
        result = operand2;
        if else_inv then result = NOT(result);
        if else_inc then result = result + 1;

    X[d] = result;
```

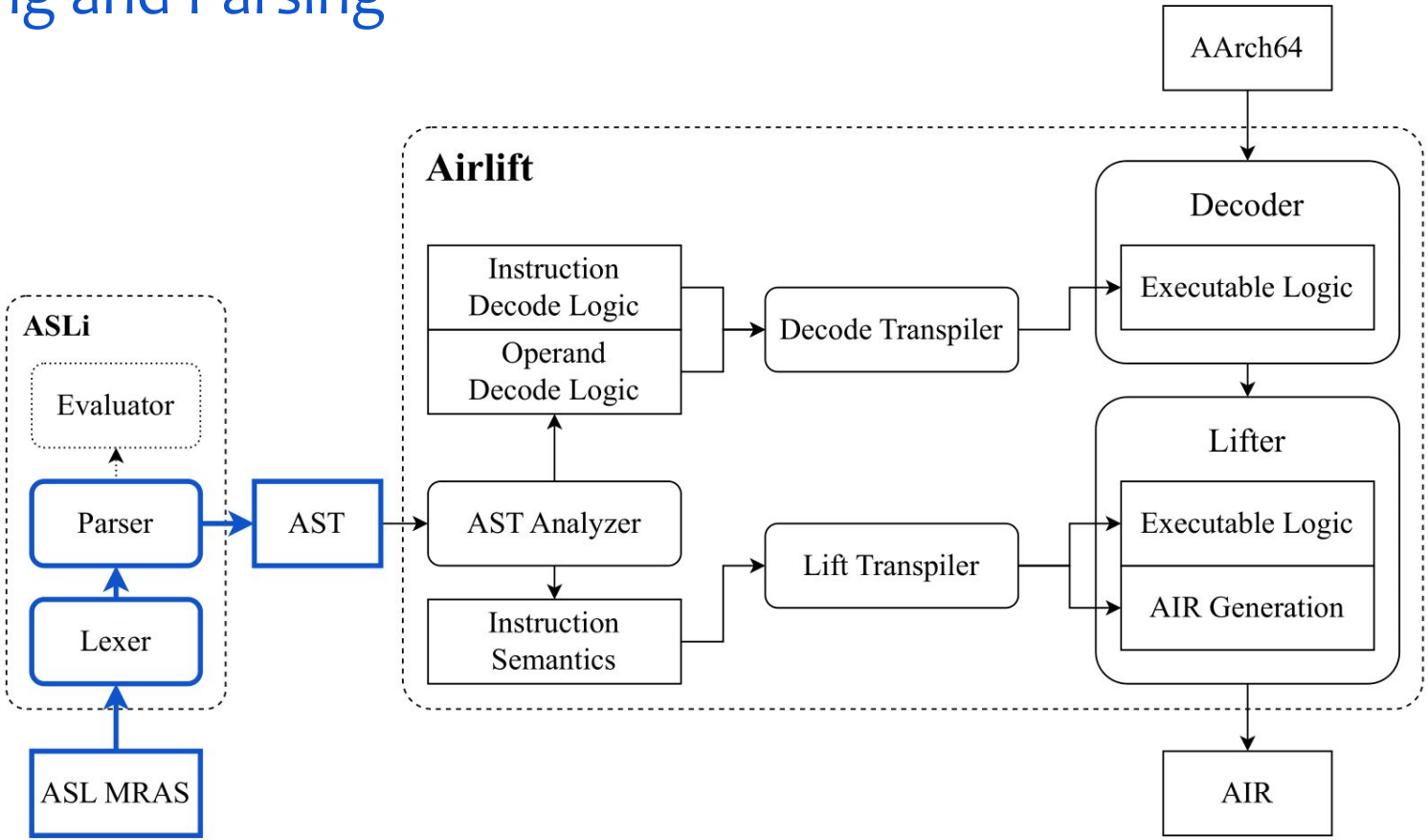
Instruction Semantics

Airlift: Workflow

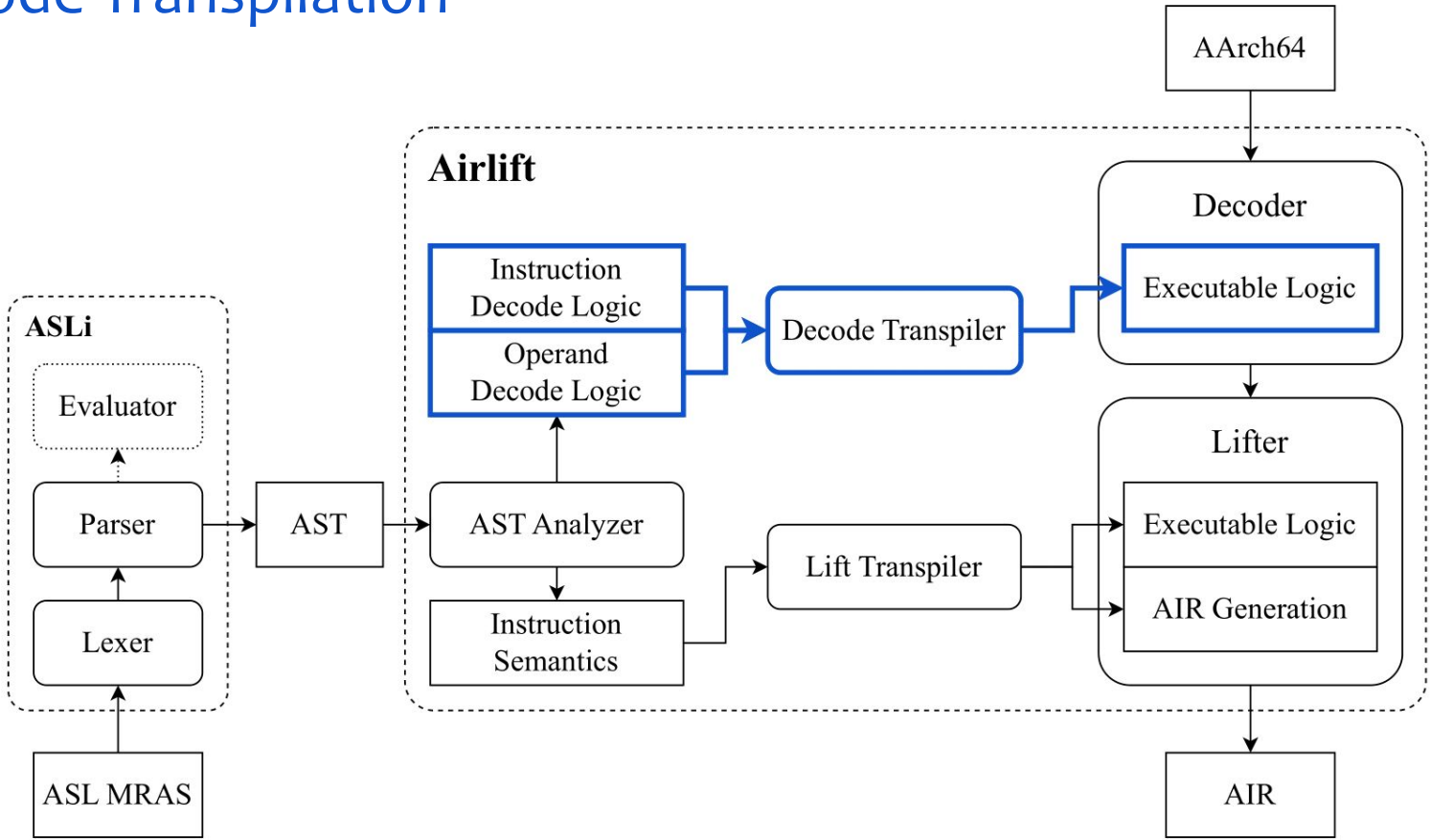
1. Codegen Runtime
2. Lifter Runtime



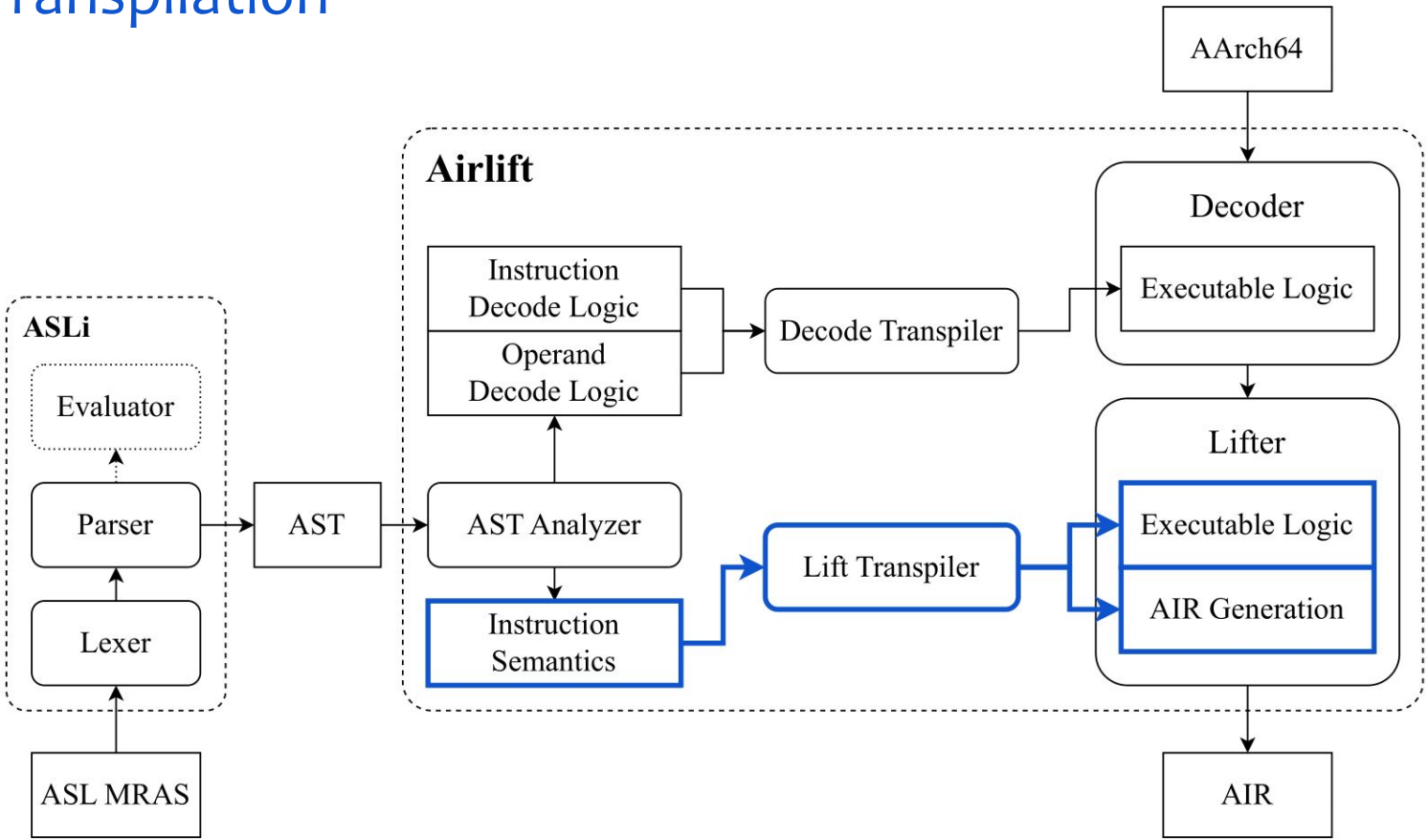
Lexing and Parsing



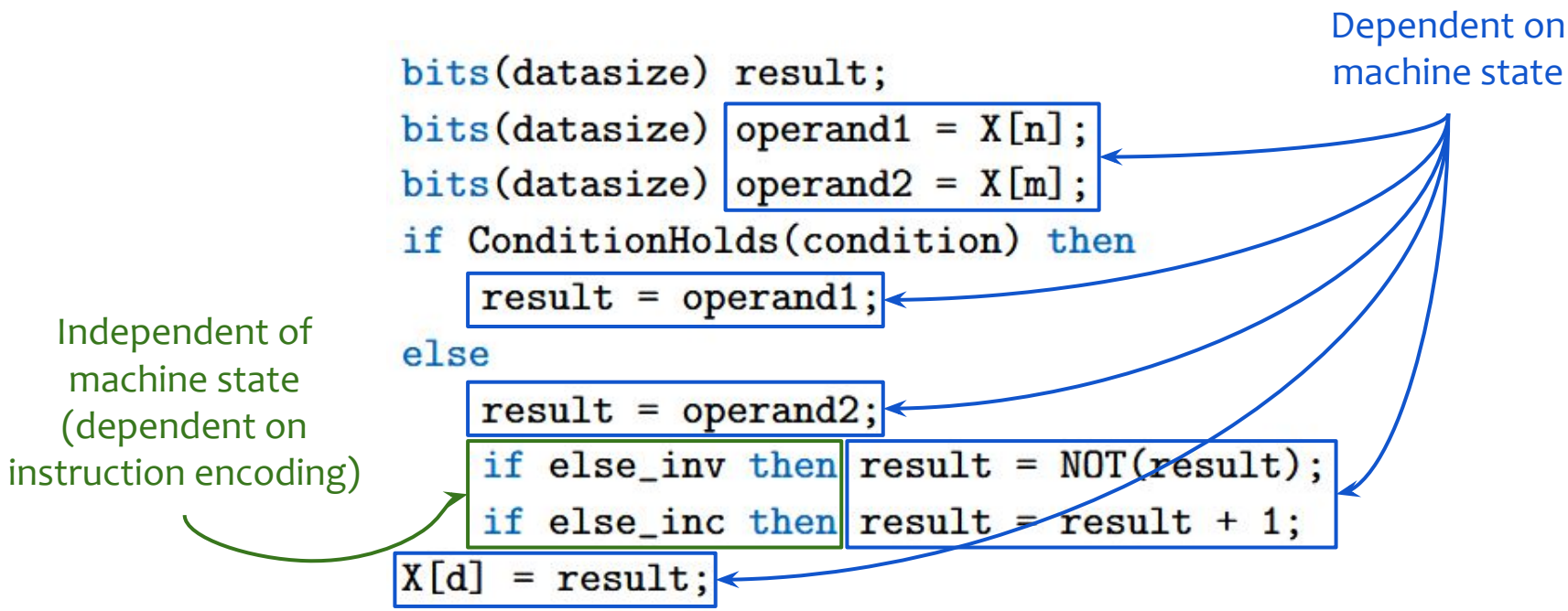
Decode Transpilation



Lift Transpilation



Partial Evaluation Example



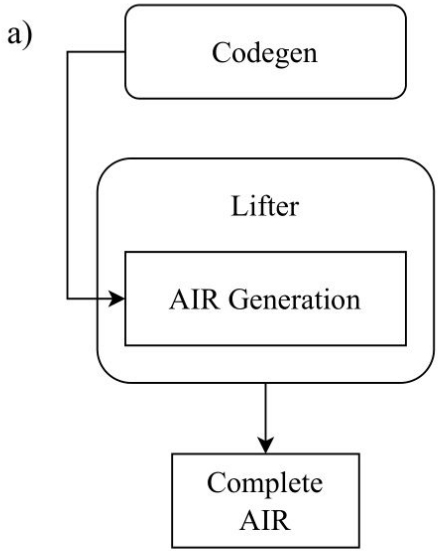
Partial Evaluation Example

```
__execute
    bits(datasize) result;
    bits(datasize) operand1 = X[n];
    bits(datasize) operand2 = X[m];
    if ConditionHolds(condition) then
        result = operand1;
    else
        result = operand2;
        if else_inv then result = NOT(result);
        if else_inc then result = result + 1;
    X[d] = result;
```

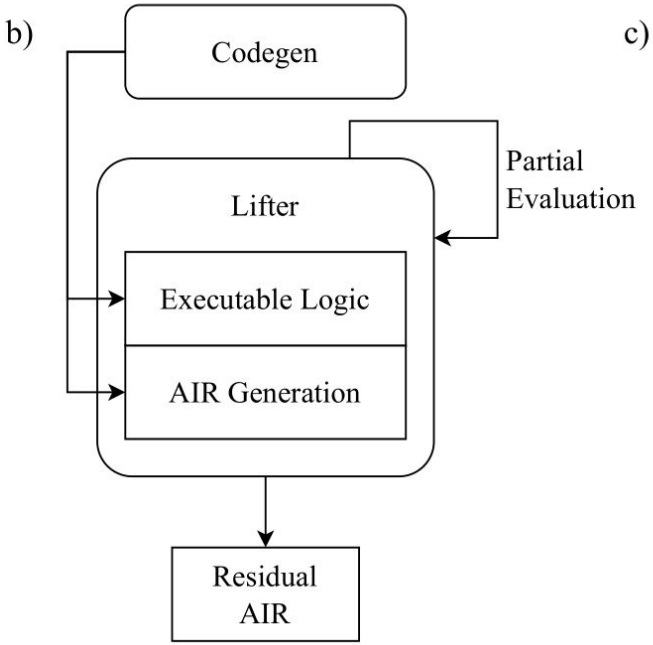
```
entry:
    v0 = i64.read_reg "x1"
    v1 = i64.read_reg "x2"
    v2 = i1.read_reg "n"
    v3 = bool.icmp.i1.eq v2, 0x1
    jumpif v3, addr_0_block_0, addr_0_block_1
addr_0_block_0:
    jump addr_0_block_2(v0)
addr_0_block_1:
    jump addr_0_block_2(v1)
addr_0_block_2(v4: i64):
    write_reg.i64 v4, "x0"
```

Partial Evaluation Alternatives

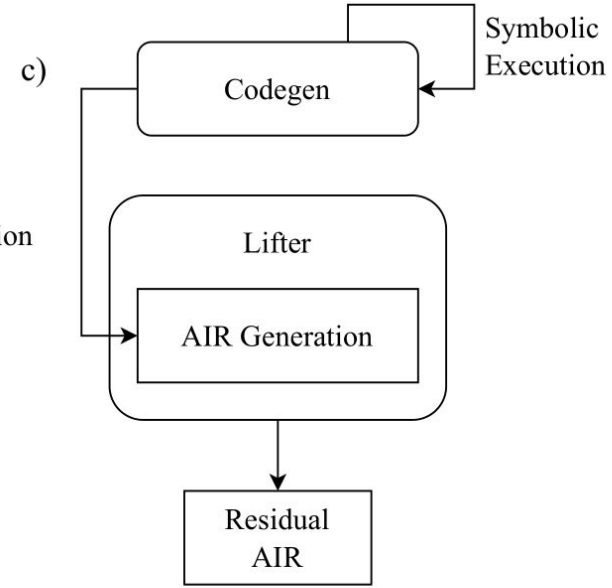
Naive



Airlift



Lift-Offline



Implementation

- ASL interpreter (OCaml + Ott)
- AST serialized into custom JSON structure
- Airlift implemented in Rust
- FP/SIMD omitted (robust fallback generated)

{JSON}

Evaluation

RQ1. Can Airlift reliably decode and lift real-world programs?

RQ2. Is its performance on real-world programs sufficient for practical use?

RQ3. What are the time, instruction count, and block count characteristics across instruction types?

RQ4. Which instruction patterns cause inefficiencies, and what insights do they offer for improvement?

Sightglass Benchmarking

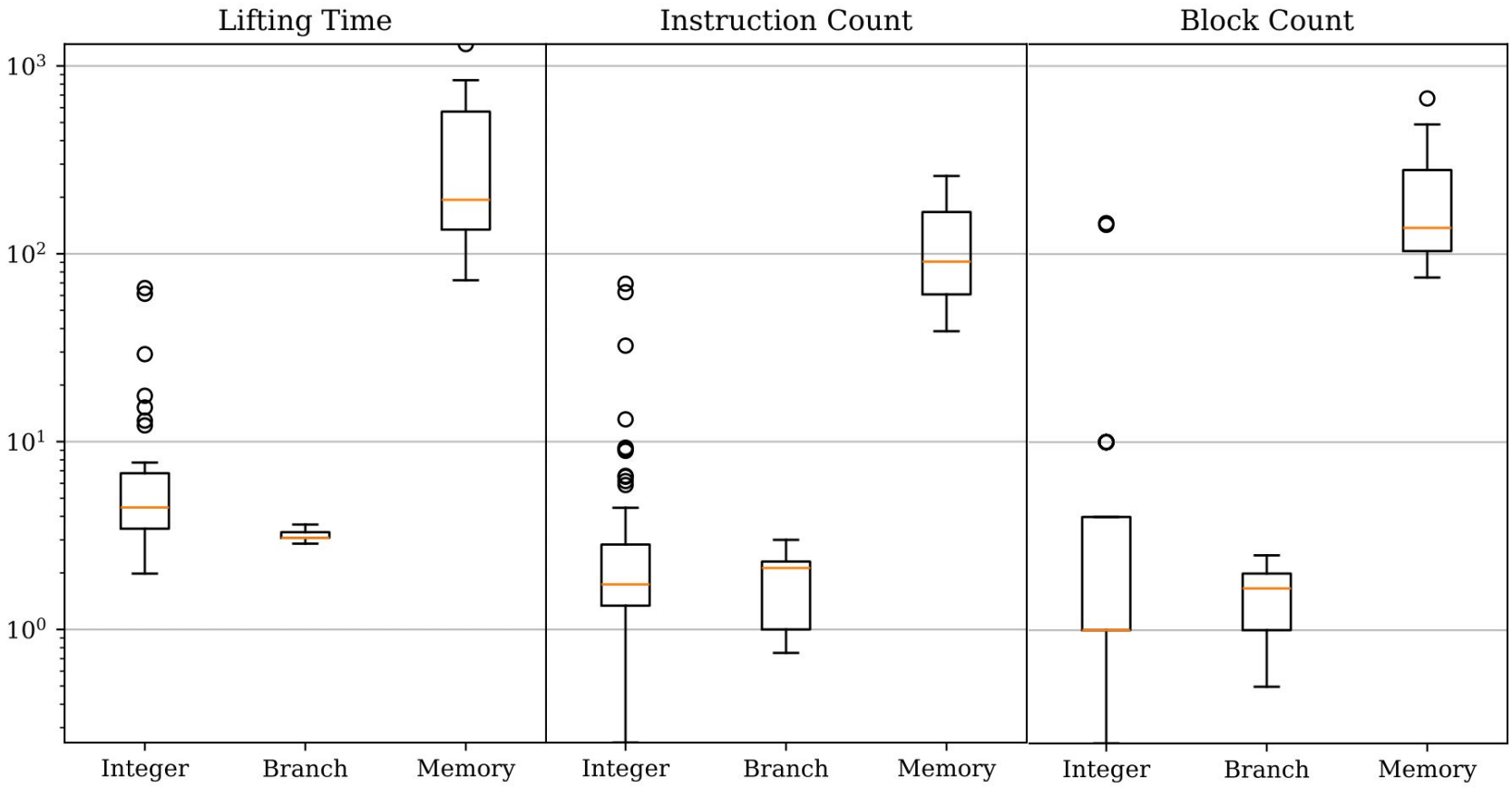
Experimental setup

- CPU: AMD EPYC 7713P 64-Core Processor
- RAM: 991 GiB
- OS: NixOS

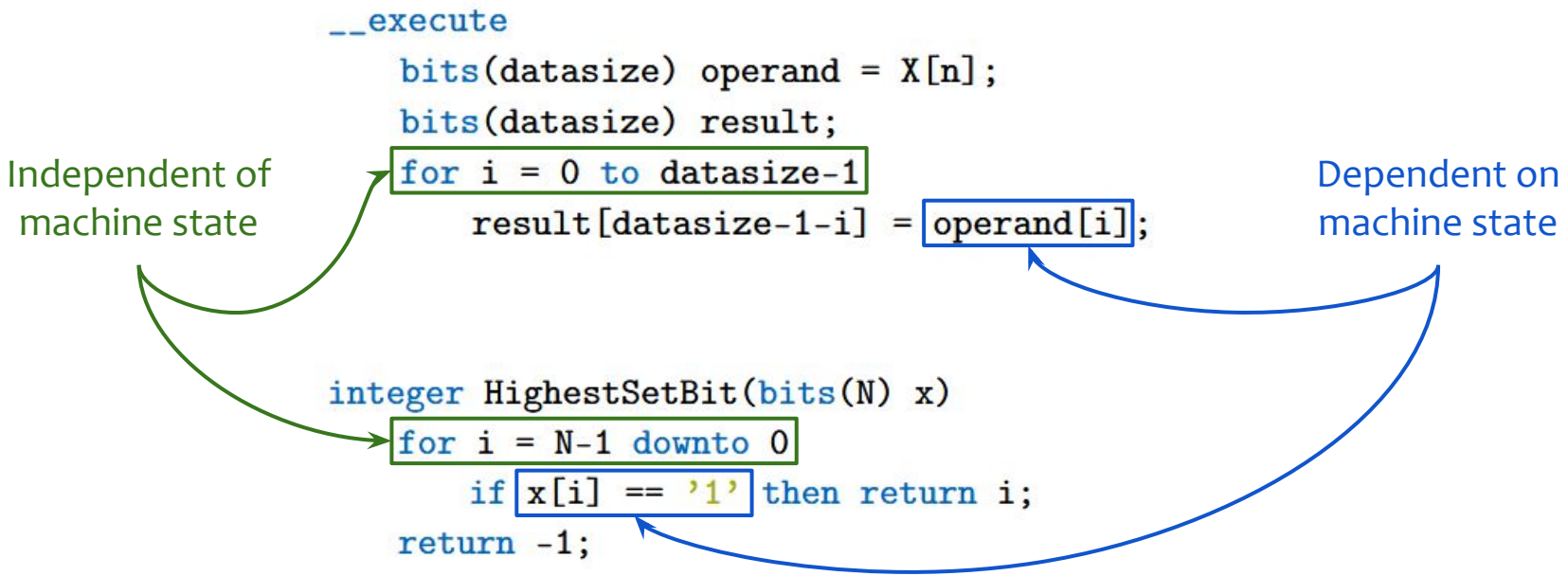
Results

- Successfully decoded all instructions in 114 Sightglass WASM applications
- Successfully lifted all instructions except the deprioritized ones, notably FP/SIMD
- 1111 times slower than `objdump`

Comparison with Manually Written Baseline Lifter



Source of Inefficiency: Loops



Summary

Airlift:

- Leverages Arm's machine-readable architecture specification
- Partially evaluates instruction semantics to reduce generated IR
- Focuses on correctness over performance

Strengths:

- Strictly adheres to formal specification
- Reliably decodes and lifts real-world applications

Limitation:

- Impractical lifting time → significant room for optimization

Try it out!

https://github.com/TUM-DSE/airlift