Optimizing Bril with STOKE

Kei Imada
Agenda

Background

STOKE

Bril

Optimizing Bril with STOKE

Overview

Multiphase Optimization

Evaluation

Future Work
Background
STOKE

Random walk on x86 assembly programs with

- Opcode (add x y → mul x y)
- Operand (add x y → add a y)
- Swap (line x ↔ line y)
- Instruction (add x y → random instr)

Accept with probability $e^{-\beta \frac{\text{cost(next program)}}{\text{cost(program)}}}$

Beta: higher= stricter acceptance

Two phases

- Synthesis: $\text{cost}(x) = \text{eq}(x), \beta = \beta_{\text{min}}$
- Optimization: $\text{cost}(x) = \text{eq}(x) + \text{perf}(x), \beta = \beta_{\text{max}}$
Bril

Language for Cornell CS6120

Inspired from LLVM IR

Statically typed

Basic arithmetic operations + other extensions

https://github.com/sampsyo/bril

```bril
# clobber.bril
@main(a: int, b: int): int {
  # (a + b) * (a + b)
  sum1: int = add a b;
  sum2: int = add a b;
  prod1: int = mul sum1 sum2;

  # Clobber both sums.
  sum1: int = const 0;
  sum2: int = const 0;

  # Use the sums again.
  sum3: int = add a b;
  prod2: int = mul sum3 sum3;

  ret prod2;
}
```
Optimizing Brill with STOKE
Optimizing with STOKE

BLOKE
# clobber.bril

@main(a: int, b: int): int {
    sum1: int = add a b;
    sum2: int = add a b;
    prod1: int = mul sum1 sum2;
    sum1: int = const 0;
    sum2: int = const 0;
    sum3: int = add a b;
    prod2: int = mul sum3 sum3;
    ret prod2;
}
# clobber.bril

@main(a: int, b: int): int {
    sum1: int = add a b;
    sum2: int = add a b;
    prod1: int = mul sum1 sum2;
    sum1: int = const 0;
    sum2: int = const 0;
    sum3: int = add a b;
    prod2: int = mul sum3 sum3;
    ret prod2;
}
# 229.73 seconds

```java
@main(a: int, b: int): int {
    sum3: int = add a b;
    prod2: int = mul sum3 sum3;
    ret prod2;
    nop;
    sum1: int = add a a;
    x4: bool = fge x2 x2;
    sum2: int = mul a sum1;
    nop;
}
```
BLOKE

STOKE on Bril

Same mutation operations

Similar cost functions

@main(a: int, b: int): int {
    sum1: int = add a b;
    sum2: int = add a b;
    prod1: int = mul sum1 sum2;
    sum1: int = const 0;
    sum2: int = const 0;
    sum3: int = add a b;
    prod2: int = mul sum3 sum3;
    ret prod2;
}

@main(a: int, b: int): int {
    nop;
    nop;
    sum3: int = add a b;
    nop;
    nop;
    nop;
    prod2: int = mul sum3 sum3;
    ret prod2;
}

STOKE Mutations

Opcode (add x y → mul x y)
Operand (add x y → add a y)
Swap (line x ↔ line y)
Instruction (add x y → random instr)
Architecture
Architecture
Architecture
Architecture
Multiphase Optimization

Recall: STOKE has two phases

Synthesis and optimization

Recall: Beta = acceptance strictness

Gamma: performance cost weight

\[ \text{cost}(x) = \text{eq}(x) + \gamma \cdot \text{perf}(x) \]

Phase 1
\[ \beta = \beta_1 = \beta_{\text{min}} \]
\[ \gamma = \gamma_1 = 0 \]

Phase 2
\[ \beta = \beta_2 \]
\[ \gamma = \gamma_2 \]

Phase n
\[ \beta = \beta_n = \beta_{\text{max}} \]
\[ \gamma = \gamma_n = 1 \]
Optimizer Architecture

Python multiprocessing + threading

Each phase is a thread

Communicate with queues
Evaluation
Benchmarks

Many existing Bril benchmarks have loops :(  
Most existing benchmarks have currently unsupported operations :(  
  Print, Call, Memory, etc  
So I made my own programs  
  Toy examples
Optimized Code Performance

![Bar chart showing the comparison between original and optimized code performance for different benchmarks.](chart.png)
Optimized Code Performance

# distribute.bril: a*b+a*c
@main(a: int, b: int, c: int): int {
    x1: int = mul a b;
    x2: int = mul a c;
    x3: int = add x1 x2;
    ret x3;
}

# dead_code.bril
@main(x: int): int {
    one: int = const 1;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    ret one;
}

# 10.68 seconds
@main(a: int, b: int, c: int): int {
    x1: int = mul a b;
    x2: int = mul a c;
    x3: int = add x1 x2;
    ret x3;
}

# 277.86 seconds
@main(x: int): int {
    one: int = const 1;
    ret one;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    jmp;
    x: int = add x one;
}
# unused_br.bril
@main(a: int, b: int): int {
    true: bool = const true;
    br true .then .else;
    .then:
        ret a;
    .else:
        ret b;
}

# dead_code_alt.bril
@main(x: int): int {
    one: int = const 1;
    y: int = id x;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    x: int = add x one;
    ret y;
}

# 418.19 seconds
@main(x: int): int {
    ret x;
    x: int = add x one;
    x: int = add x one;
    x3: bool = fle x0 x1;
    y: int = id x;
    x: int = mul x x;
    x: int = sub x x;
    x5: bool = lt x one;
    ret y;
}

# 228.13 seconds
@main(a: int, b: int): int {
    ret a;
    br true .then .else;
    .then:
        true: bool = const true;
    .else:
        ret b;
}

# 418.19 seconds
@main(x: int): int {
    ret x;
    x: int = add x one;
    x: int = add x one;
    x3: bool = fle x0 x1;
    y: int = id x;
    x: int = mul x x;
    x: int = sub x x;
    x5: bool = lt x one;
    ret y;
Multiphase Optimized Code Performance

Number of Phases Impact on Optimized Code

- distribute
- dead_code
- clobber
- unused_br
- dead_code_alt

Total Dynamic Instructions Executed vs Number of Phases
Multiphase Performance

Number of Phases Impact on Runtime

- distribute
- dead_code
- clobber
- unused_br
- dead_code_alt

Runtime (seconds)

Number of Phases
Scalability

Process Count Impact on Runtime

- distribute
- dead_code
- clobber
- unused_br
- dead_code_alt
Future Work

More benchmarks
Support more Bril operations in Z3
  Print statements with theory of arrays
  Pointers and memory access
  Function calls
  Speculation
Trace optimizations and JIT compilation
Allow some loops with loop-invariant synthesis
Optimizer optimizations
MPI for distributed BLOKE
Backup
Sample

def sample(program: Program) → Program:
    next_program ← mutated program
    return next_program with probability $e^{-\beta \frac{\text{cost}(\text{next_program})}{\text{cost}(\text{program})}}$
    otherwise return program

Recall: beta is the acceptance strictness
Equivalence cost

Lift loop-free programs to Z3

Program x not equivalent to $x_0$ iff $\exists i [x(i) \neq x_0(i)]$

Z3 is expensive and running brilirs on test cases is cheaper

Counter-example guided equivalence cost function

$$eq(x) = \begin{cases} \text{validation}(x) & \text{if } \text{validation}(x) > 0 \\ \text{verification}(x) & \text{otherwise} \end{cases}$$
Performance cost

Bril interpreters have `-p` flag to profile

Total dynamic instructions executed

If interpreter produces an error, use approximate performance

$$\text{total\_dyn\_inst}_{\text{approx}}(x, i) = \text{len}(x[\text{insts}]) - x[\text{number of nops}]$$

$$\text{total\_dyn\_inst}(x, i) = \begin{cases} \infty & \text{if } x \text{ has a loop} \\ \text{total\_dyn\_inst}_{\text{approx}}(x) & \text{if } x \text{ errors on input } i \\ x(i)[\text{total\_dyn\_inst}] & \text{otherwise} \end{cases}$$

$$\text{perf}(x) = \max_i \{\text{total\_dyn\_inst}(x, i)\}$$
Bril2Z3

Static single assignment (SSA) form

Bril types

- Integer: 64-bit bitvector
- Float: 64-bit floating point

Z3 datatypes to model Bril return types
Bril Modifications

Python bindings for brilirs

Allow “main” function to return an integer

    Treat as return code

brili and brilirs no longer profiles nops