UFMG

Don’t Care Analysis for Verilog Designs

Compilers Laboratory

Rafael Paniago
rafaelpradopaniago@gmail.com

Consider this basic verilog function

```
module foo (
     input [31:0] a, b, c,
     output [31:0] result
);
     wire [31:0] mult = a * b;
     assign result = mult + c;

endmodule
```

Consider this basic verilog function

```
module foo (
     input [31:0] a, b, c,
     output [31:0] result
);
     wire [31:0] mult = a * b;
     assign result = mult + c;

endmodule
```



32 bit signal

This is what it looks like in C

```
module foo (
    input [31:0] a, b, c,
    output [31:0] result
);
    wire [31:0] mult = a * b;
    assign result = mult + c;

endmodule
```

```
void foo(
    int a, int b, int c,
    int* result)
{
    *result = a * b + c;
}
```

Could something be improved in this code?

DCC

```
module foo (
    input [31:0] a, b, c,
    output [31:0] result
);
    wire [31:0] mult = a * b;
    assign result = mult + c;

endmodule
```

```
module bar (
    input [31:0] in1, in2, in3,
    output [15:0] out
);
    wire [31:0] full_result;
    foo my_instance (
        .a(in1),
        .b(in2),
        .c(in3),
        .result(full_result)
    );
assign out = full_result[15:0];
endmodule
```

Could something be improved in this code?

```
module foo (
    input [31:0] a, b, c,
    output [31:0] result
);
    wire [31:0] mult = a * b;
    assign result = mult + c;

endmodule
```

Module Instantiation



```
module bar (
    input [31:0] in1, in2, in3,
    output [15:0] out
);
    wire [31:0] full_result;
    foo my_instance (
        .a(in1),
        .b(in2),
        .c(in3),
        .result(full_result)
    );
assign out = full_result[15:0];
endmodule
```

Could something be improved in this code?

DCC

```
module foo (
    input [31:0] a, b, c,
    output [31:0] result
);
    wire [31:0] mult = a * b;
    assign result = mult + c;

endmodule
```

```
module bar (
    input [31:0] in1, in2, in3,
    output [15:0] out
);
    wire [31:0] full_result;
    foo my_instance (
        .a(in1),
        .b(in2),
        .c(in3),
        .result(full_result)
    );
assign out = full_result[15:0];
endmodule
```

Bits [31:16] are unused!

```
module foo (
    input [31:0] a, b, c,
    output [31:0] result
);
    wire [31:0] mult = a * b;
    assign result = mult + c;

endmodule
```

```
module bar (
    input [31:0] in1, in2, in3,
    output [15:0] out
);
    wire [31:0] full_result;
    foo my_instance (
        .a(in1),
        .b(in2),
        .c(in3),
        .result(full_result)
    );
assign out = full_result[15:0];
endmodule
```

This code would be semantically equivalent

DCC

```
module foo (
     input [15:0] a, b, c,
     output [15:0] result
);
     wire [15:0] mult = a * b;
     assign result = mult + c;

endmodule
```

```
module bar (
     input [31:0] in1, in2, in3,
     output [15:0] out
);
     wire [15:0] full_result;
     foo my_instance (
          .a(in1[15:0]),
          .b(in2[15:0]),
          .c(in3[15:0]),
          .result(full_result)
     );
assign out = full_result[15:0];
endmodule
```

What are the advantages of the second design?

DCC

What are the advantages of the second design?

DCC

- Cleaner

What are the advantages of the second design?

DCC

- Cleaner
- Lower Symbolic Complexity

What are the advantages of the second design?

DCC

- Cleaner
- Lower Symbolic Complexity
- More efficient synthesized hardware

What are our objectives?

What are our objectives?

DCC

- Create a static analysis that identifies useless bits

What are our objectives?

DCC

- Create a static analysis that identifies useless bits
- Create an optimization pass using the analysis

How do we accomplish the first objective?

DCC

How do we accomplish the first objective?

DCC

- Backwards Data Flow Analysis

How do we accomplish the first objective?

DCC

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks

How do we accomplish the first objective?

DCC

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks
- Meet operation is bitwise OR

How do we accomplish the first objective?

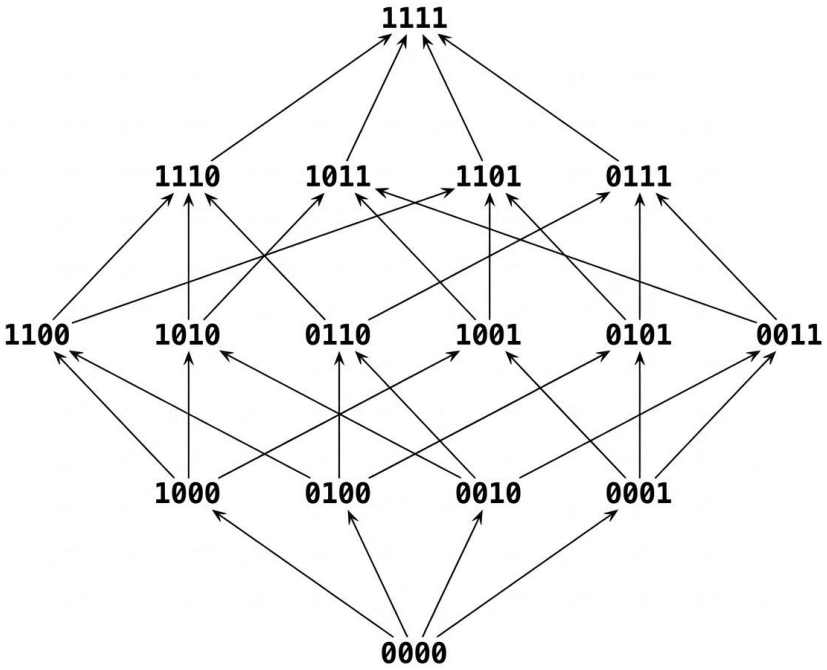
DCC

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks
- Meet operation is bitwise OR
- Outputs are set to fully observable

How do we accomplish the first objective?

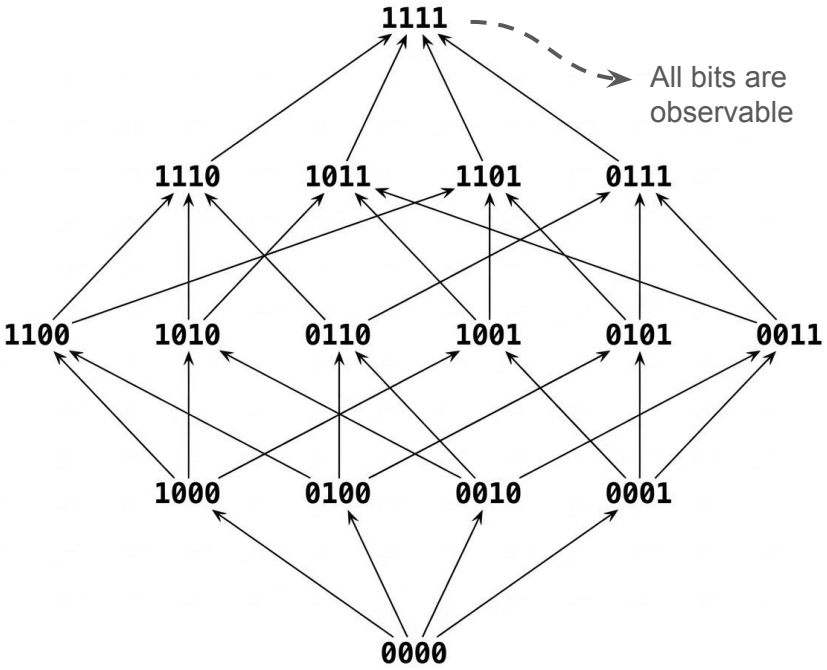
DCC

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks
- Meet operation is bitwise OR
- Outputs are set to fully observable



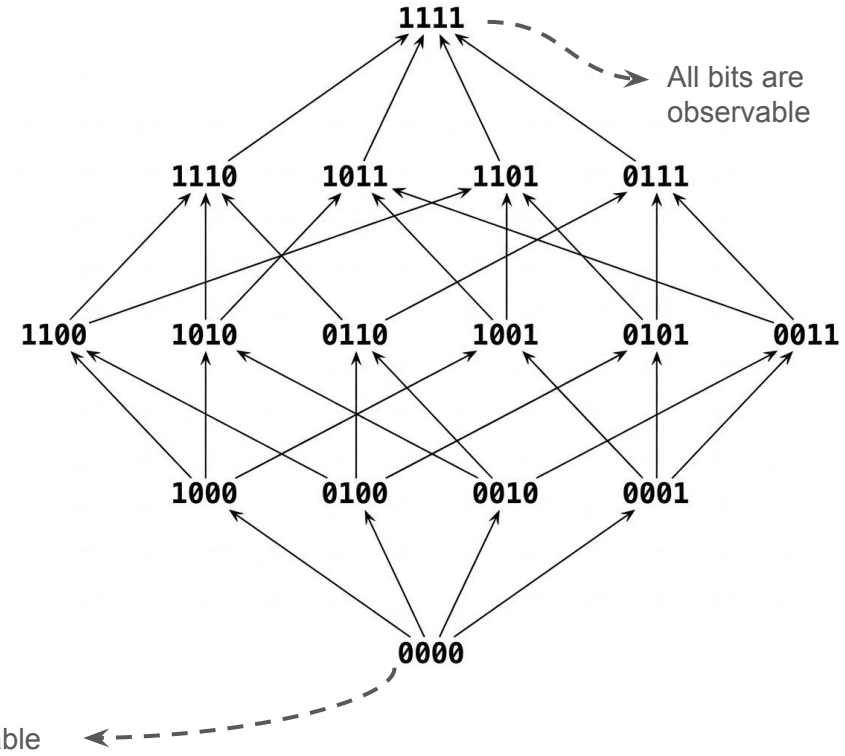
How do we accomplish the first objective?

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks
- Meet operation is bitwise OR
- Outputs are set to fully observable



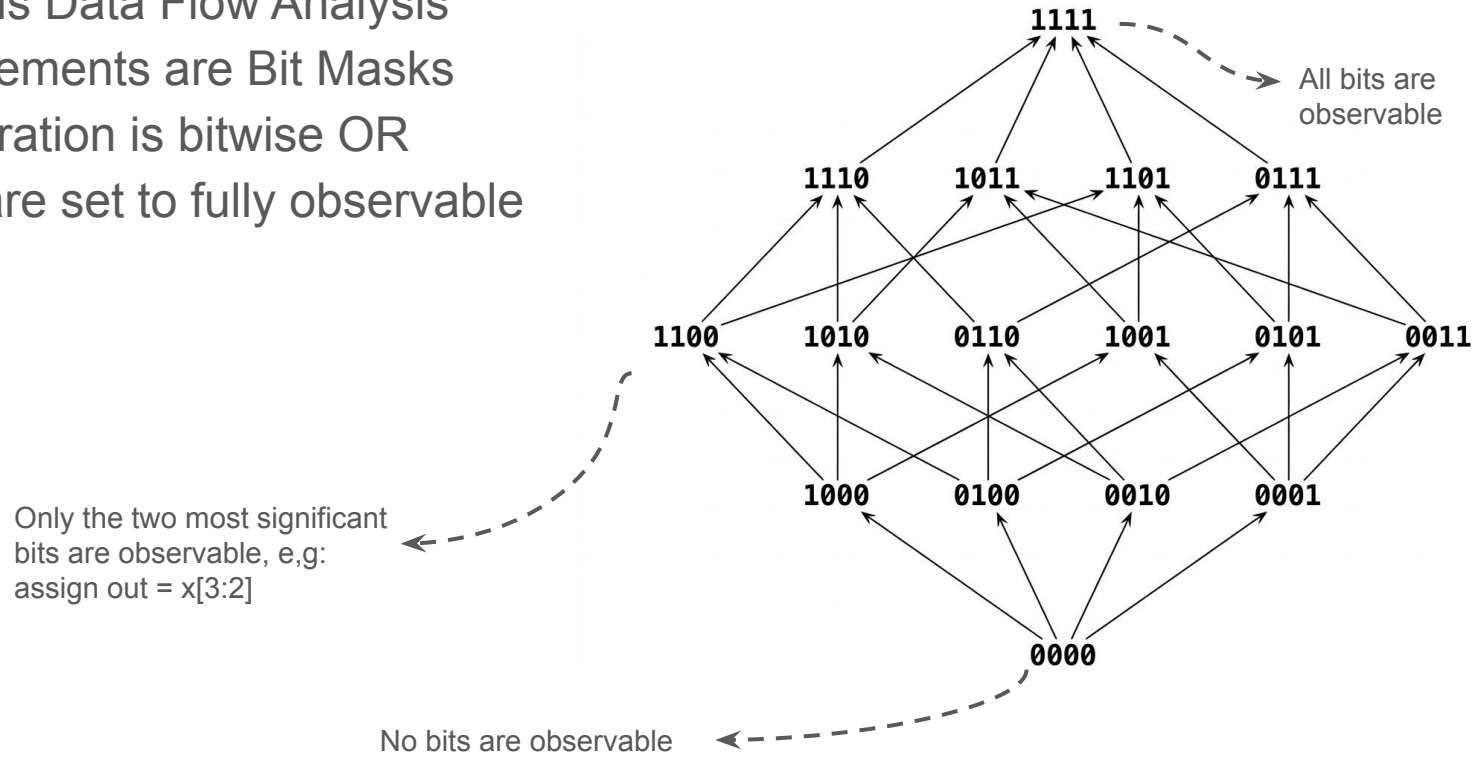
How do we accomplish the first objective?

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks
- Meet operation is bitwise OR
- Outputs are set to fully observable



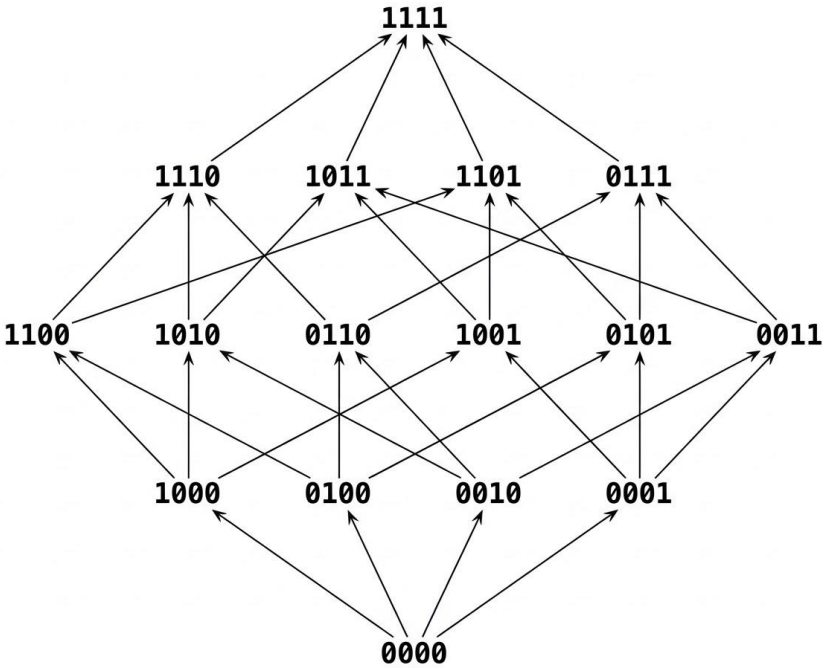
How do we accomplish the first objective?

- Backwards Data Flow Analysis
- Lattice Elements are Bit Masks
- Meet operation is bitwise OR
- Outputs are set to fully observable



What about the second one?

```
wire [31:0] y;
assign y[1:0] = x[1:0];
assign out = y[1:0] ;
```

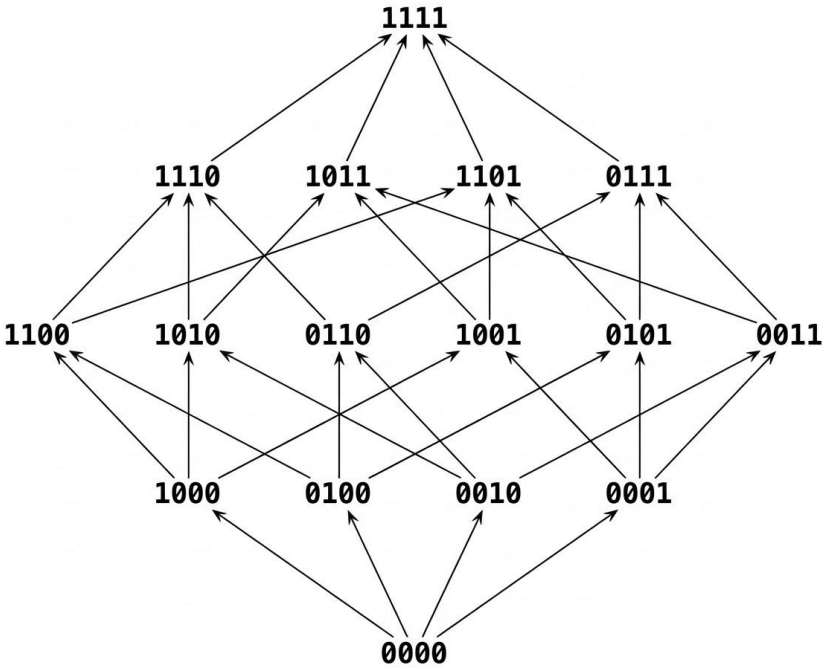


What about the second one?

- We apply masks to the values

```
wire [31:0] y;
assign y[1:0] = x[1:0];
assign out = y[1:0] ;
```

```
wire [31:0] y;
assign y = {30'h0, x[1:0]}
assign out = y[1:0] ;
```



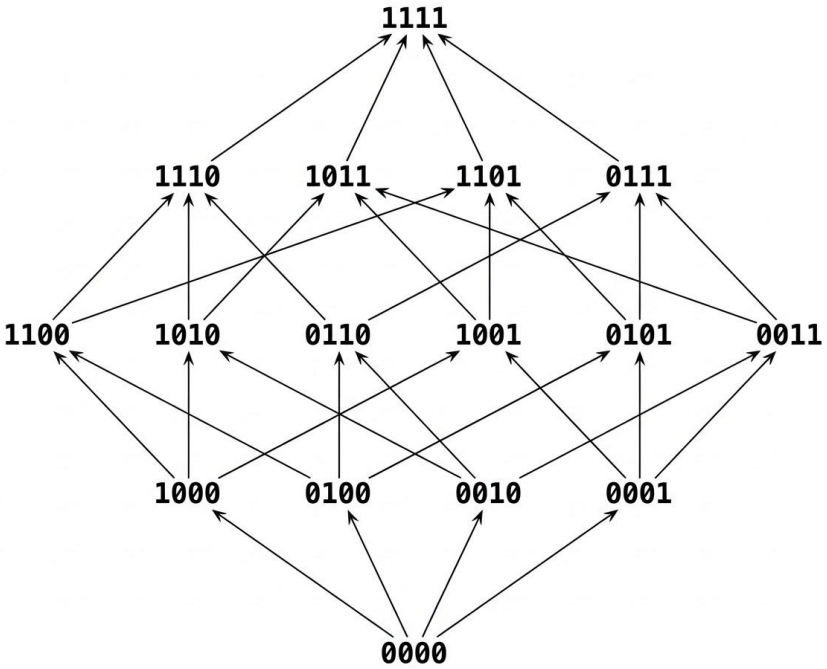
What about the second one?

- We apply masks to the values
- And let firtool do the rest

```
wire [31:0] y;
assign y[1:0] = x[1:0];
assign out = y[1:0] ;
```

```
wire [31:0] y;
assign y = {30'h0, x[1:0]}
assign out = y[1:0] ;
```

```
wire [1:0] y;
assign y = x;
assign out = y ;
```



CIRCT: Circuit IR Compilers and Tools

DCC



https://github.com/llvm/circt

CIRCT Dialects

DCC



https://github.com/llvm/circt

CIRCT Dialects

DCC

- Comb



https://github.com/llvm/circt

CIRCT Dialects

DCC

- Comb
- HW



https://github.com/llvm/circt

CIRCT Dialects

DCC

- Comb
- HW
- Seq



https://github.com/llvm/circt

CIRCT Dialects

DCC

- Comb
- HW
- Seq
- Sv



https://github.com/llvm/circt

CIRCT Dialects

DCC

- **Comb** 

- **HW**
- Seq
- Sv

```
hw.module @SimpleALU(in %a: i32, in %b: i32, in %sel: i1, out
result: i32) {
    %sum = comb.add %a, %b : i32
    %and_val = comb.and %a, %b : i32
    %out = comb.mux %sel, %sum, %and_val : i32
    output port hw.output %out : i32
}
```

Testing

DCC

Testing

DCC

- Formal Verification by Jasper



Testing

DCC

- Formal Verification by Jasper
- Analysis is performed before standard FIRTOOL optimizations



Current Results

Current Results

DCC

- 2% of 1000 largest ChiBench designs had identified useless bits

Current Results

DCC

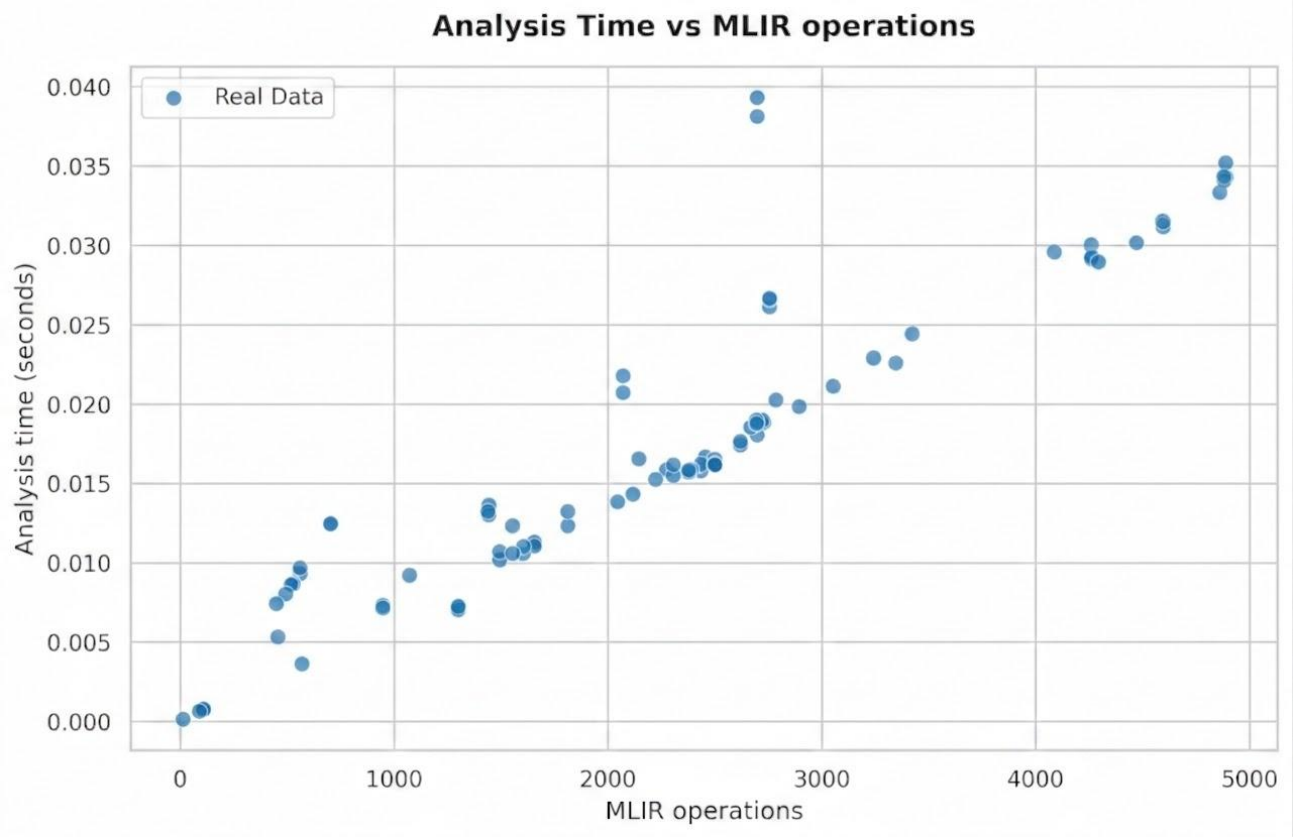
- 2% of 1000 largest ChiBench designs had identified useless bits
- Mean analysis time in 100 largest designs: 15 milliseconds

Current Results

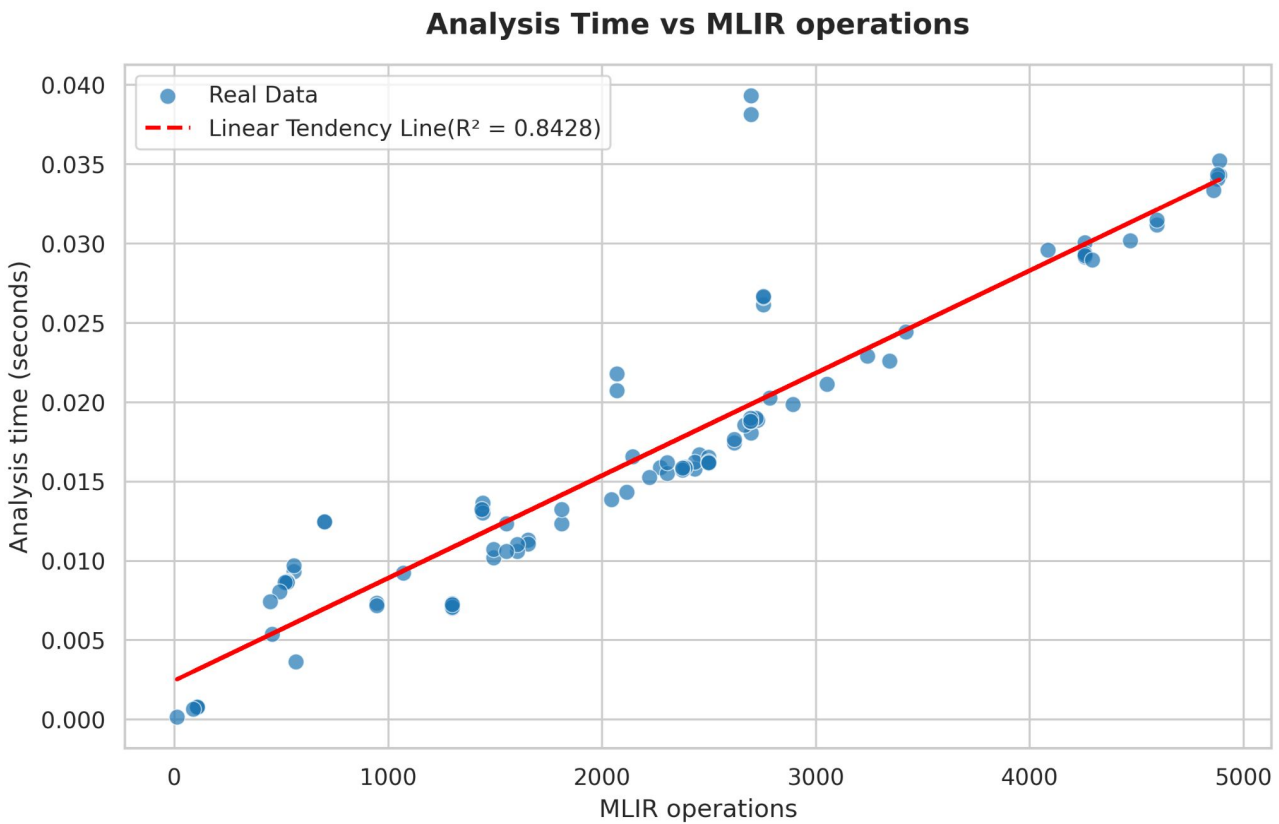
DCC

- 2% of 1000 largest ChiBench designs had identified useless bits
- Mean analysis time in 100 largest designs: 15 milliseconds
- Analysis time scales linearly with MLIR operations

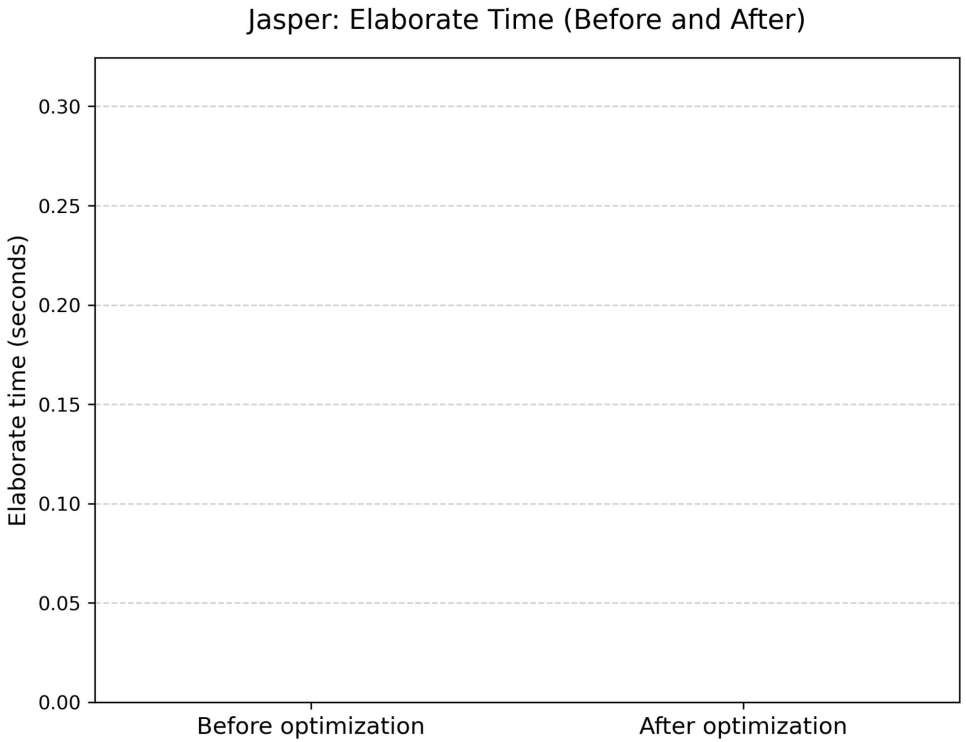
Current Results: Time complexity



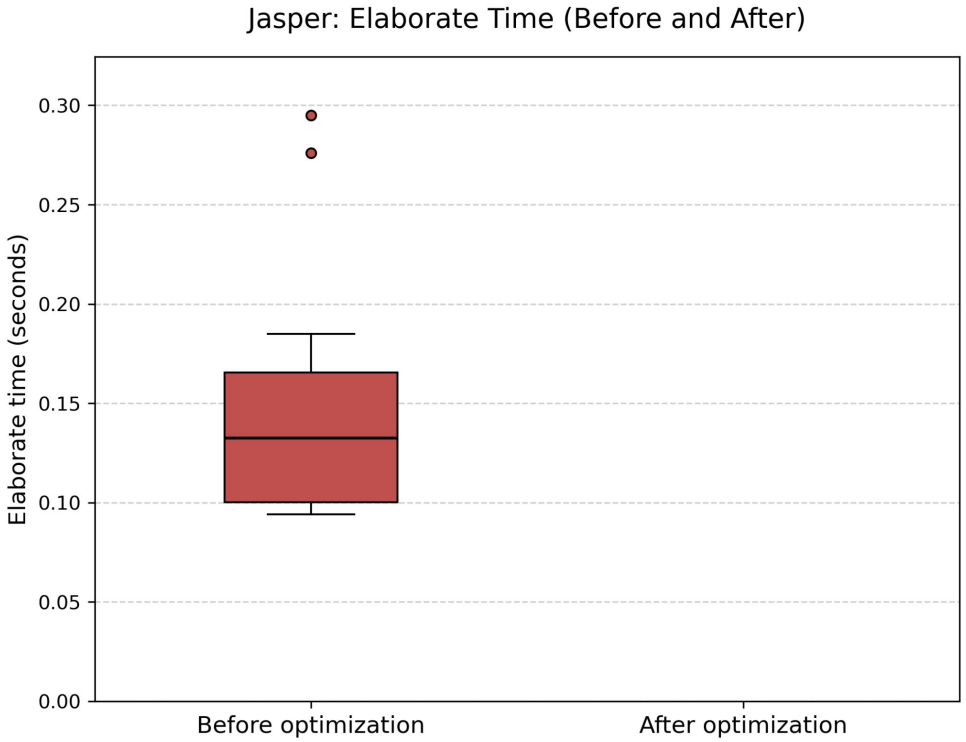
Current Results: Time complexity



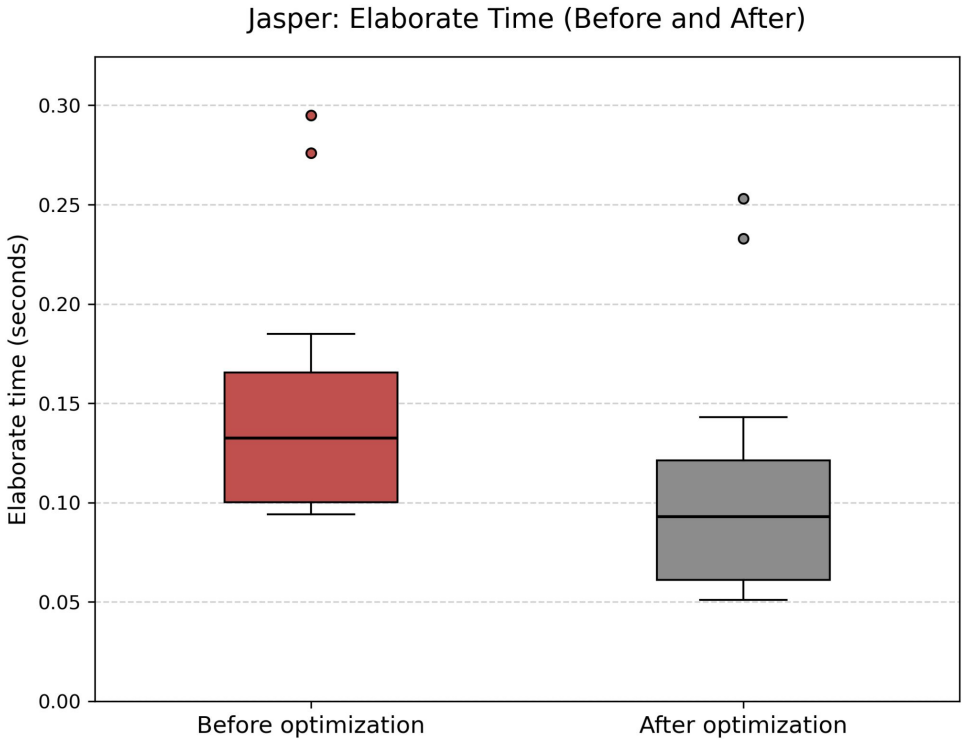
Current Results: Elaborate Times



Current Results: Elaborate Times



Current Results: Elaborate Times 30% decrease



Future Objectives

Future Objectives

DCC

- Extend transfer functions to seq and sv dialects

Future Objectives

DCC

- Extend transfer functions to seq and sv dialects
- Upstream the analysis and the pass to CIRCT

References & Contact

DCC

Rafael Paniago

rafaelpradopaniago@gmail.com

http://lac.dcc.ufmg.br