LORAIN

A Step Closer to the PDES “Holy Grail”

Justin LaPre, Elsa Gonsiorowski, Christopher Carothers
Rensselaer Polytechnic Institute

“Holy Grail”

- Automatic event handler generation
- PADS 1994, Fujimoto claimed the “Holy Grail” out of reach during that century
- New technologies may bring this within our grasp

Rensselaer's Optimistic Simulation System

- Time Warp
- Uses *reverse computation*
- Runs on millions of cores — see Barnes et al. (PADS 2013)
- Go get your own copy! github.com/carothersc/ROSS

Motivation

- Writing reverse code is hard!
- Example:

```
// Forward
if (msg->val == 1) {
    SWAP(lp->val, msg->val);
}

// Reverse
if (msg->val == 1) {
    SWAP(lp->val, msg->val);
}
```

Looks “obviously” correct forward and backward, but it’s not.

Motivation

- Writing reverse code is hard!
- Example: swapping into messages is common!

```
// Forward
if (msg->val == 1) {
    SWAP(lp->val, msg->val);
}

// Reverse
if (msg->val == 1) {
    SWAP(lp->val, msg->val);
}
```

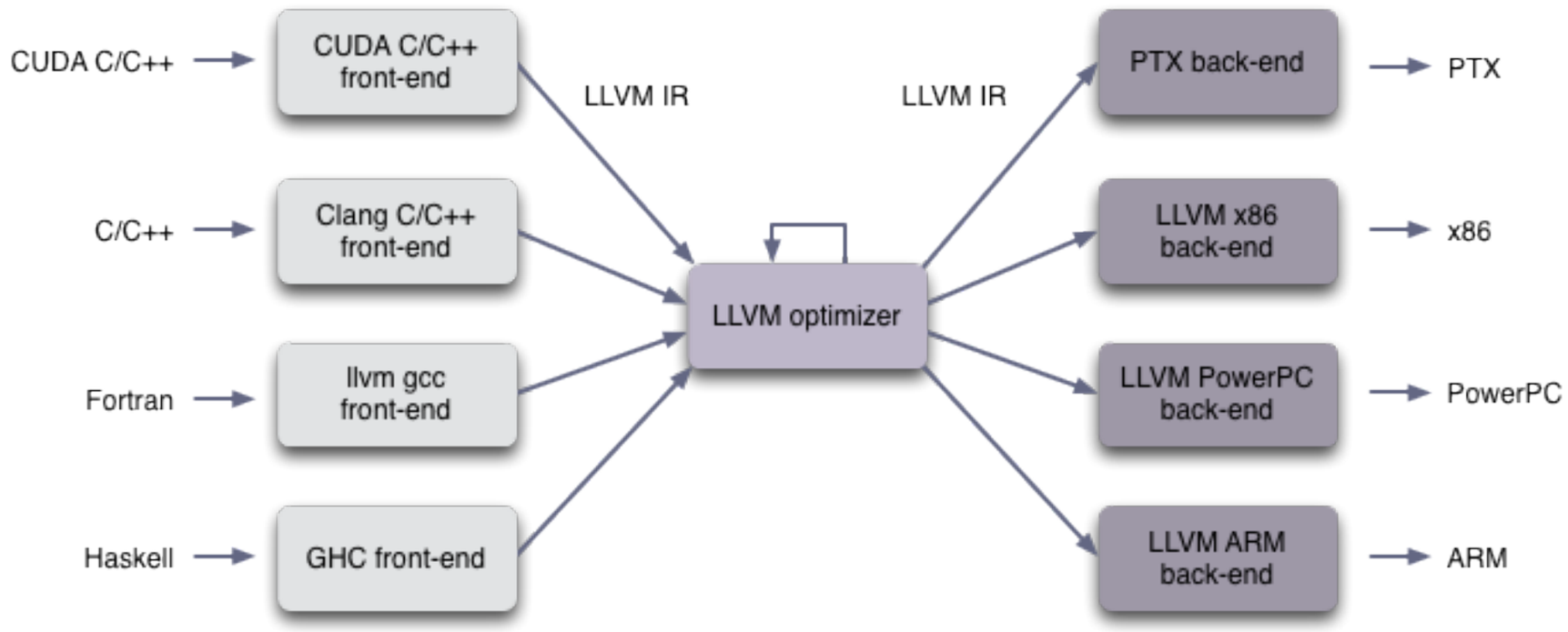
Looks “obviously” correct forward and backward, but it’s not.

```
// Forward
if (msg->val == 1) {
    SWAP(lp->val, msg->val);
}

// Reverse
if (lp->val == 1) {
    SWAP(lp->val, msg->val);
}
```

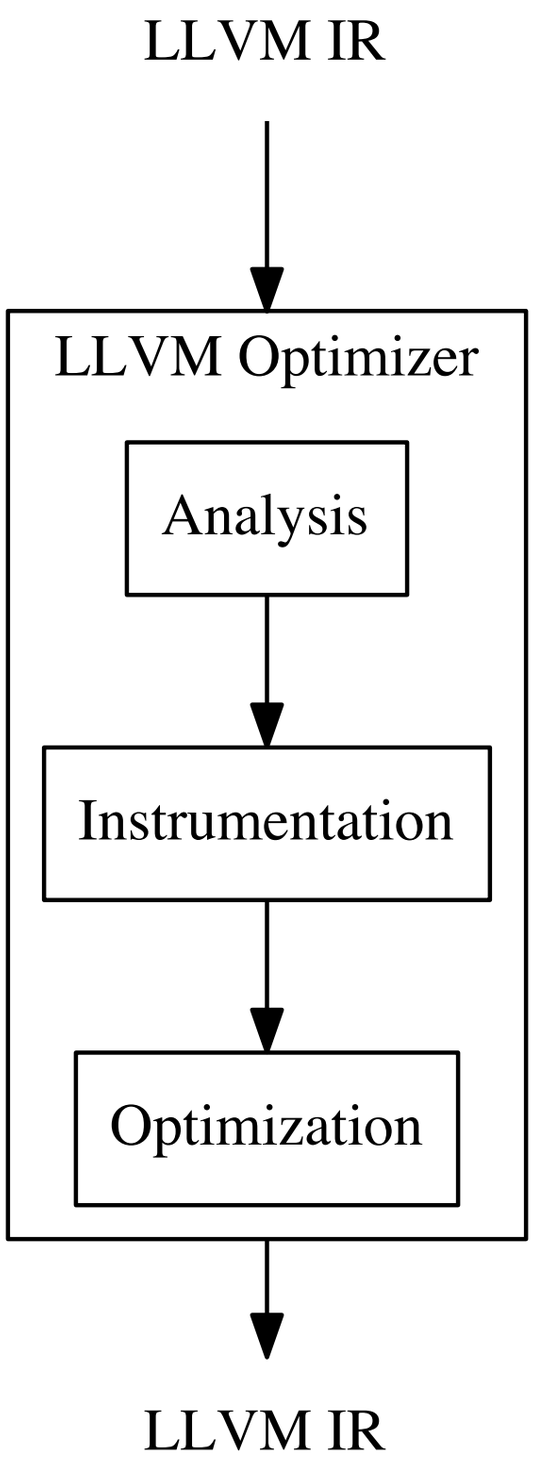
Hard error to spot!

LLVM

- Fast, modular, well-designed compiler
- Multiple front-ends, multiple back-ends
- Operates on IR



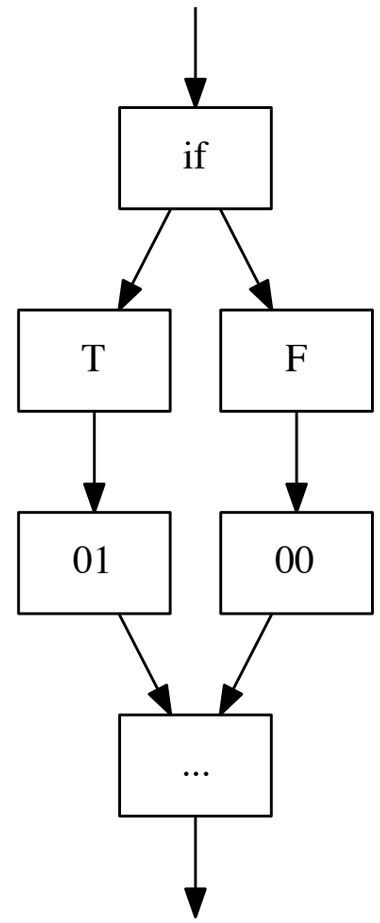
Optimizer


LLVM IR
LLVM Optimizer
Analysis
Instrumentation
Optimization
LLVM IR

What Does LLVM IR Look Like?

```
void test_if(void)
{
    if (test_if_x_condition) {
        test_if_x = test_if_x + 1;
    }
}
```

```
define void @test_if() nounwind uwtable ssp {
  %1 = load i32* @test_if_x_condition, align 4
  %2 = icmp ne i32 %1, 0
  br i1 %2, label %3, label %6

; <label>:3     ; preds = %0
  %4 = load i32* @test_if_x, align 4
  %5 = add nsw i32 %4, 1
  store i32 %5, i32* @test_if_x, align 4
  br label %6

; <label>:6     ; preds = %3, %0
  ret void
}
```



Our Approach

- How does the model change state? **Store Instructions**
- Keep track of path (so we can retrace it)
- Ignore everything else

3 Steps

- Augment the appropriate “message” data structure to support swap operations
  - Catch “destructive” operations e.g., `x = 5`
- Instrument / analyze the forward event handler
  - Mark non-important `store` instructions with metadata
- Clone and invert the forward event handler

Stores

Constructive	Destructive
int x = 1;	int x = 5;
x = x + 1;	x = 7;

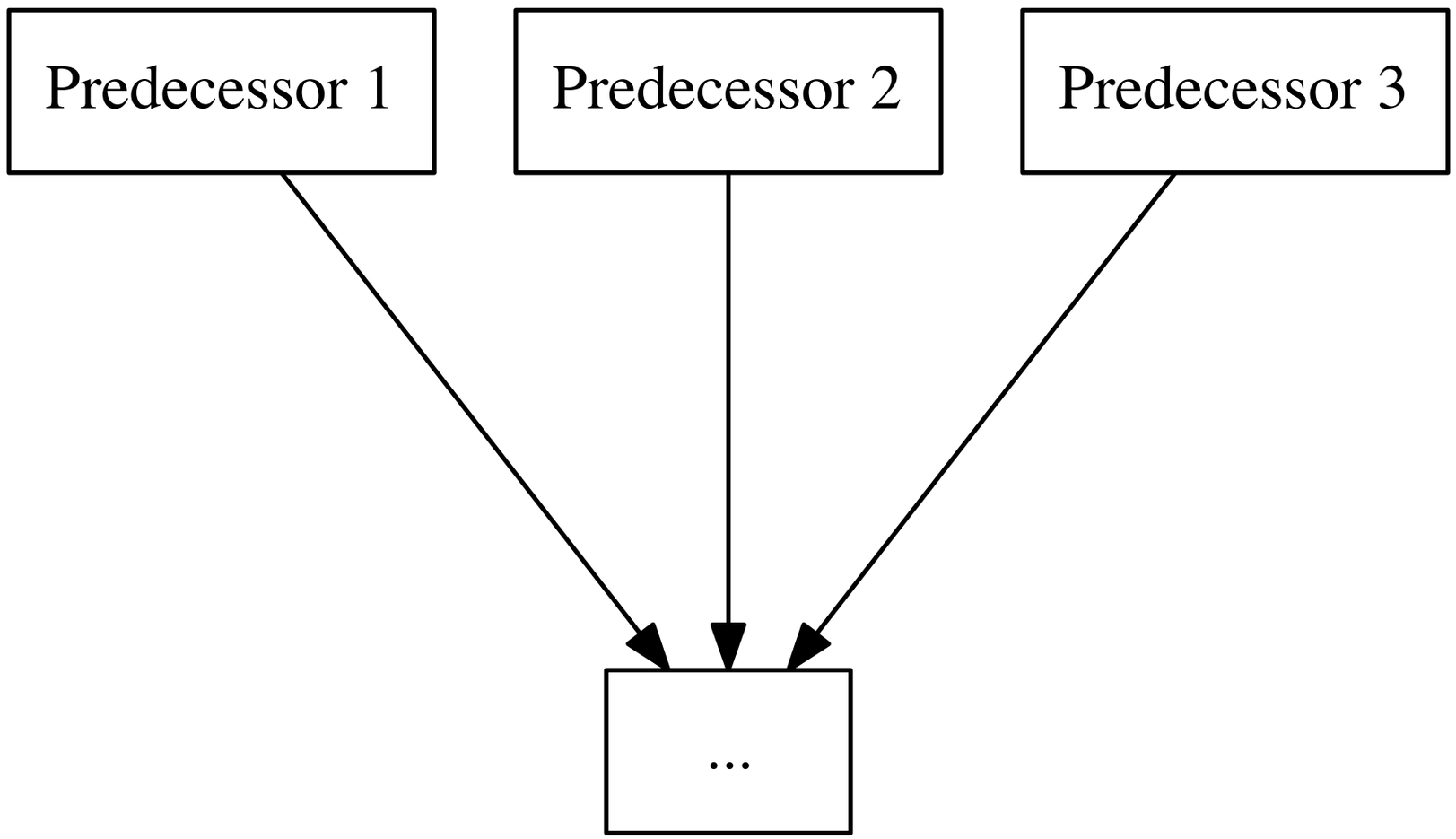
Message Augmentation

```
struct msg {

  int var1;

  double var2;

}
```

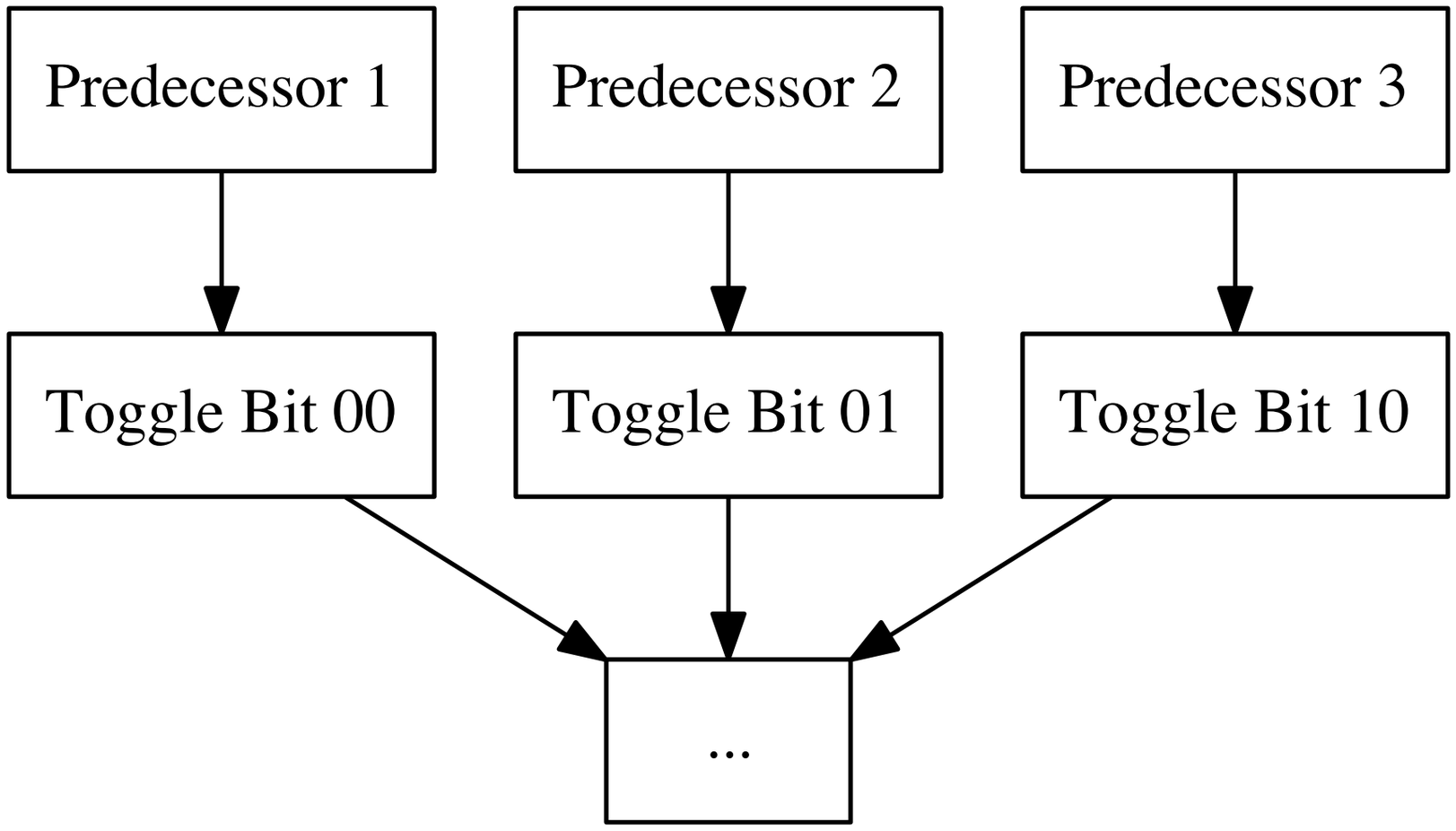
Message Augmentation

```
struct msg.augmented {
    int var1;
    double var2;
    int destroyed_var1;
}
```

Basic Block Graph Instrumentation



Basic Block Graph Instrumentation



Inversion

- Find forward *F* and reverse declaration *R*
- Clone all basic blocks from *F* to *R*
- Place remaining `store` instructions on a stack
  - Build dependencies from *use-def* chain
- Don’t forget about random numbers!

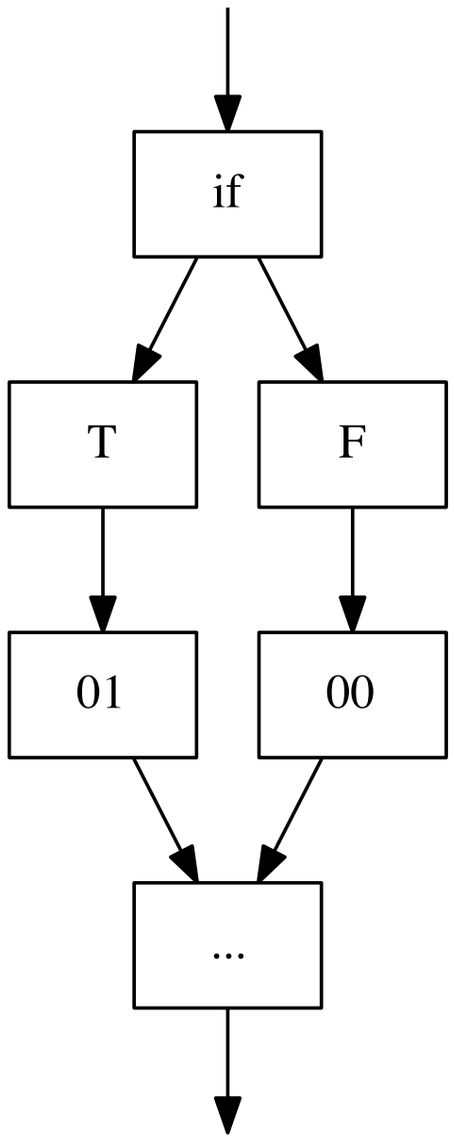
Constructive Stores

- SSA makes use-def chains easy!
- Topologically-sorted dependencies
- Addition becomes subtraction
- Don’t construct values that aren’t required

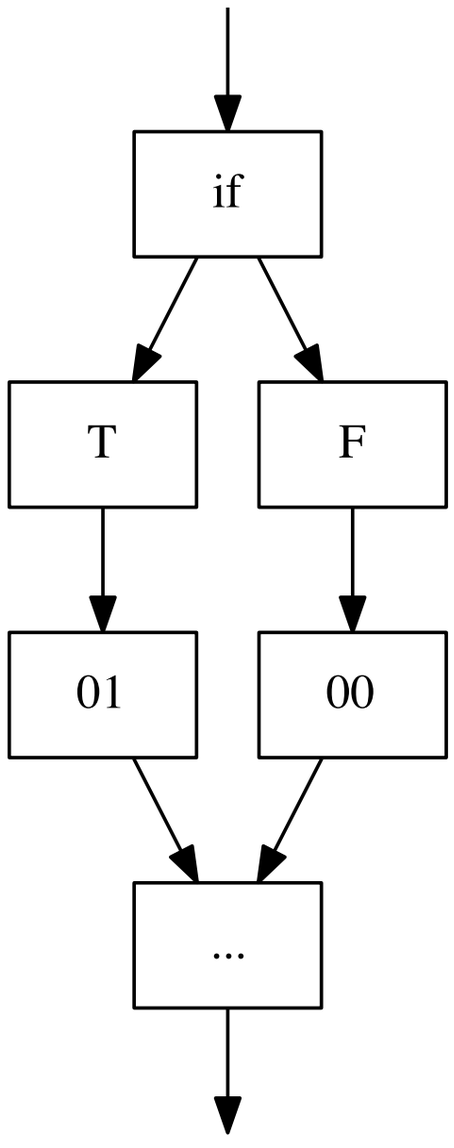
Terminator Instructions

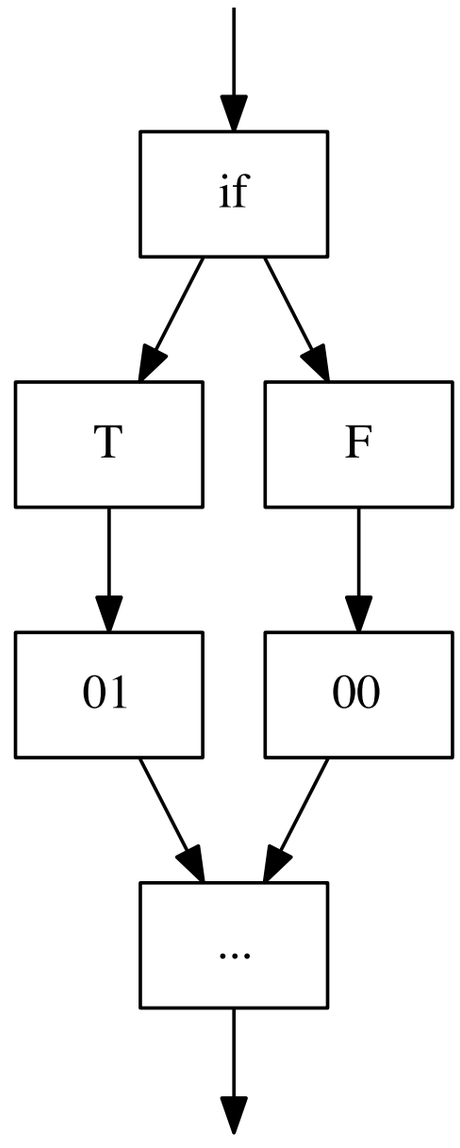
- Terminators include:
  - Branches (e.g., `if` statements)
  - Function exit (e.g., `return` or `exit`)
- From Terminators alone, reconstruct reverse-CFG
- Insert `switch` to jump to predecessor blocks

Forward

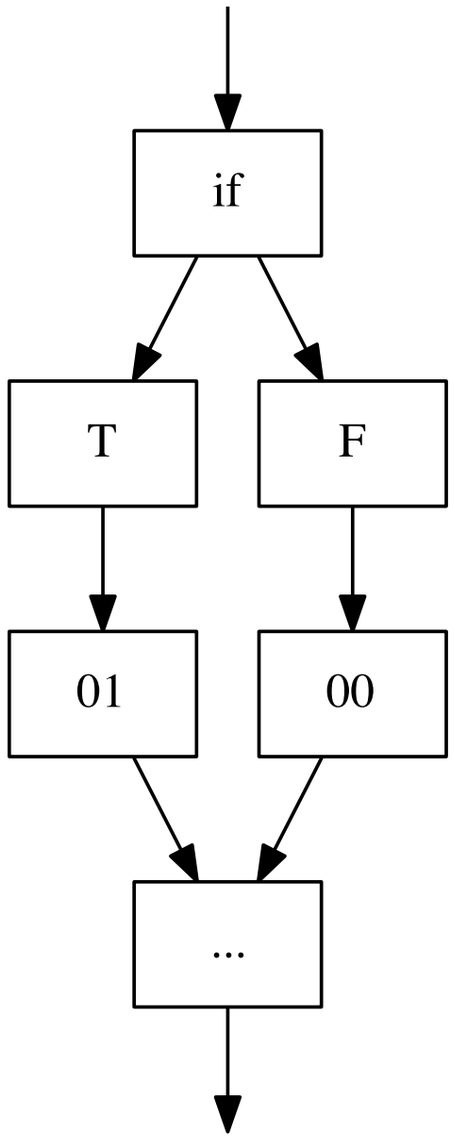


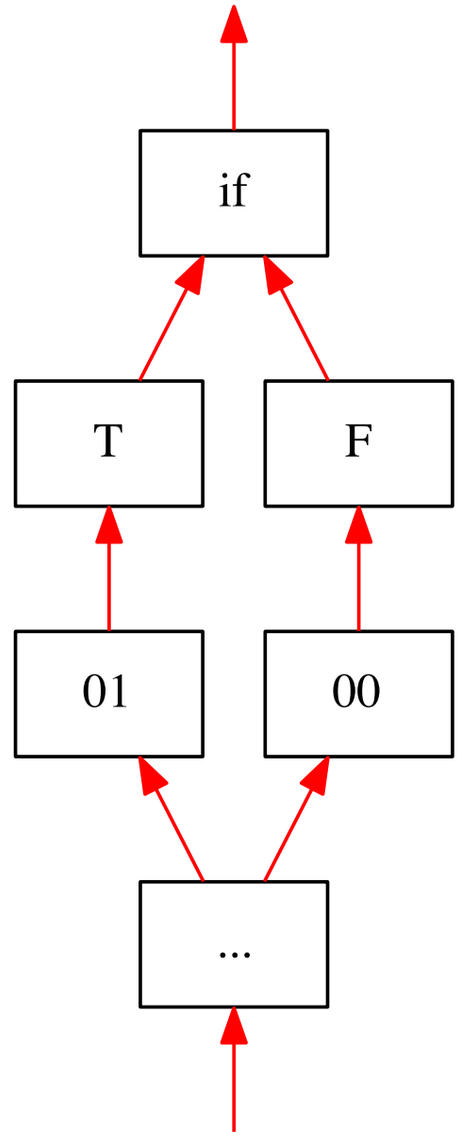
Forward and



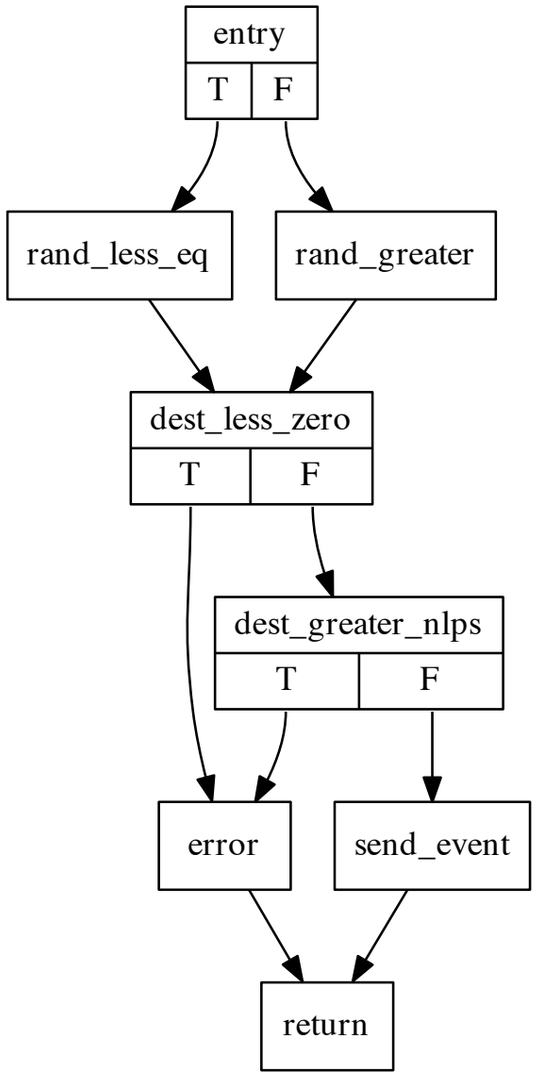


Forward and Reverse



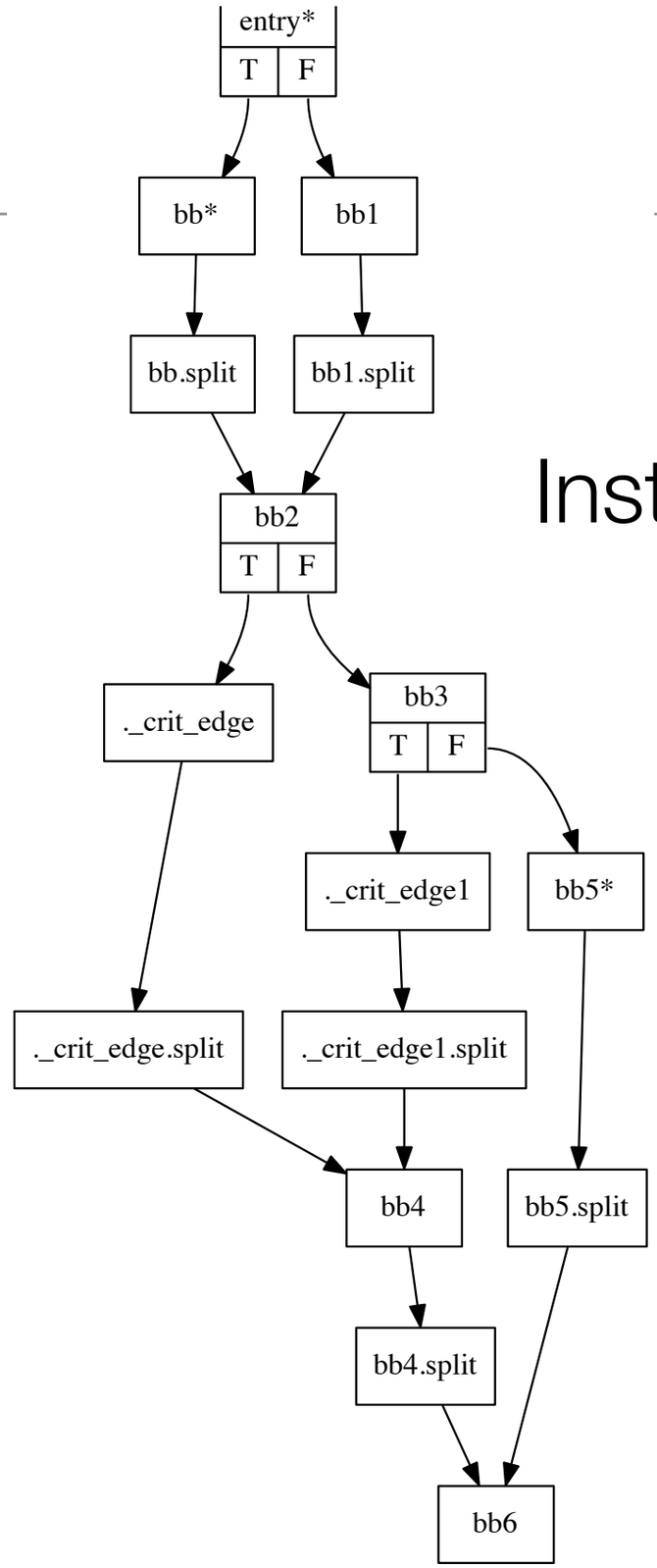


Example: phold model



CFG for 'phold_event_handler' function

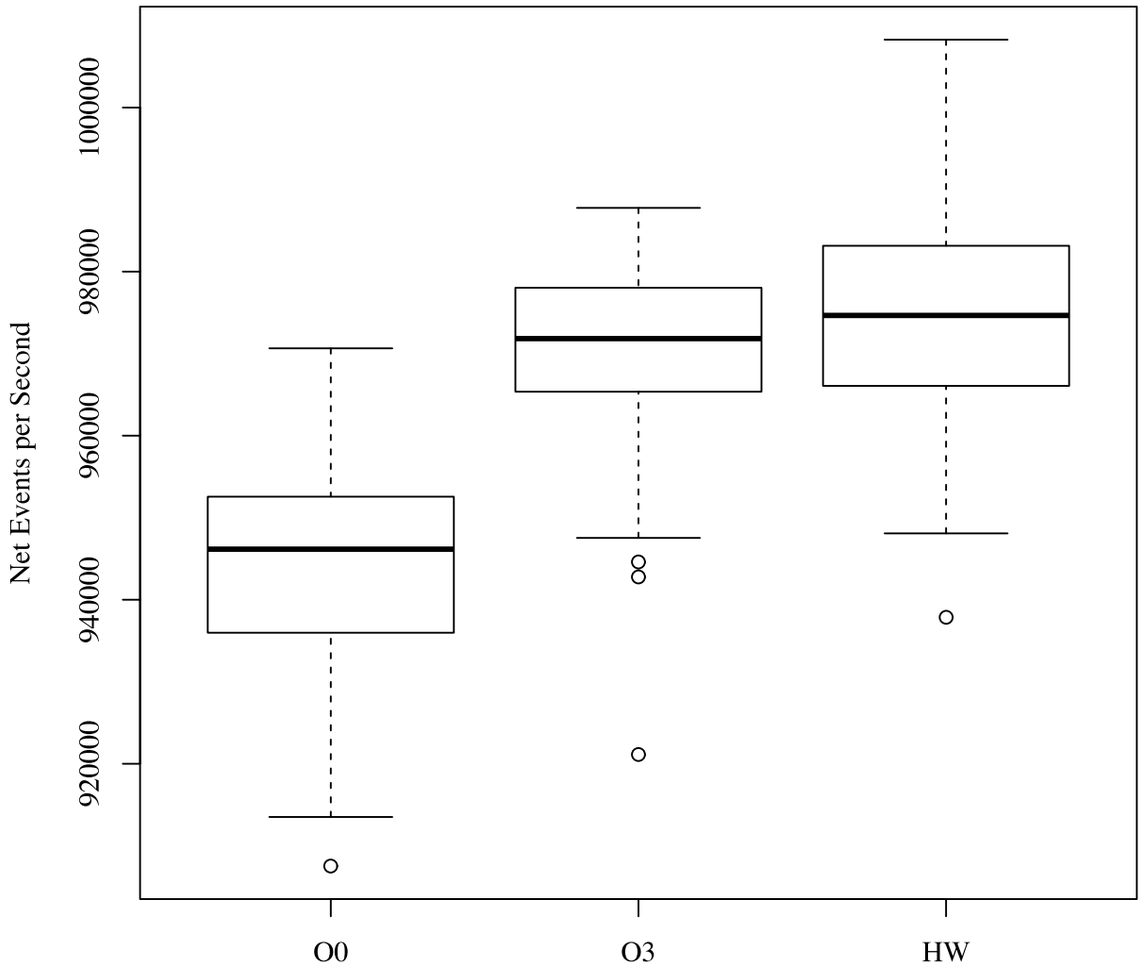
Non-instrumented!

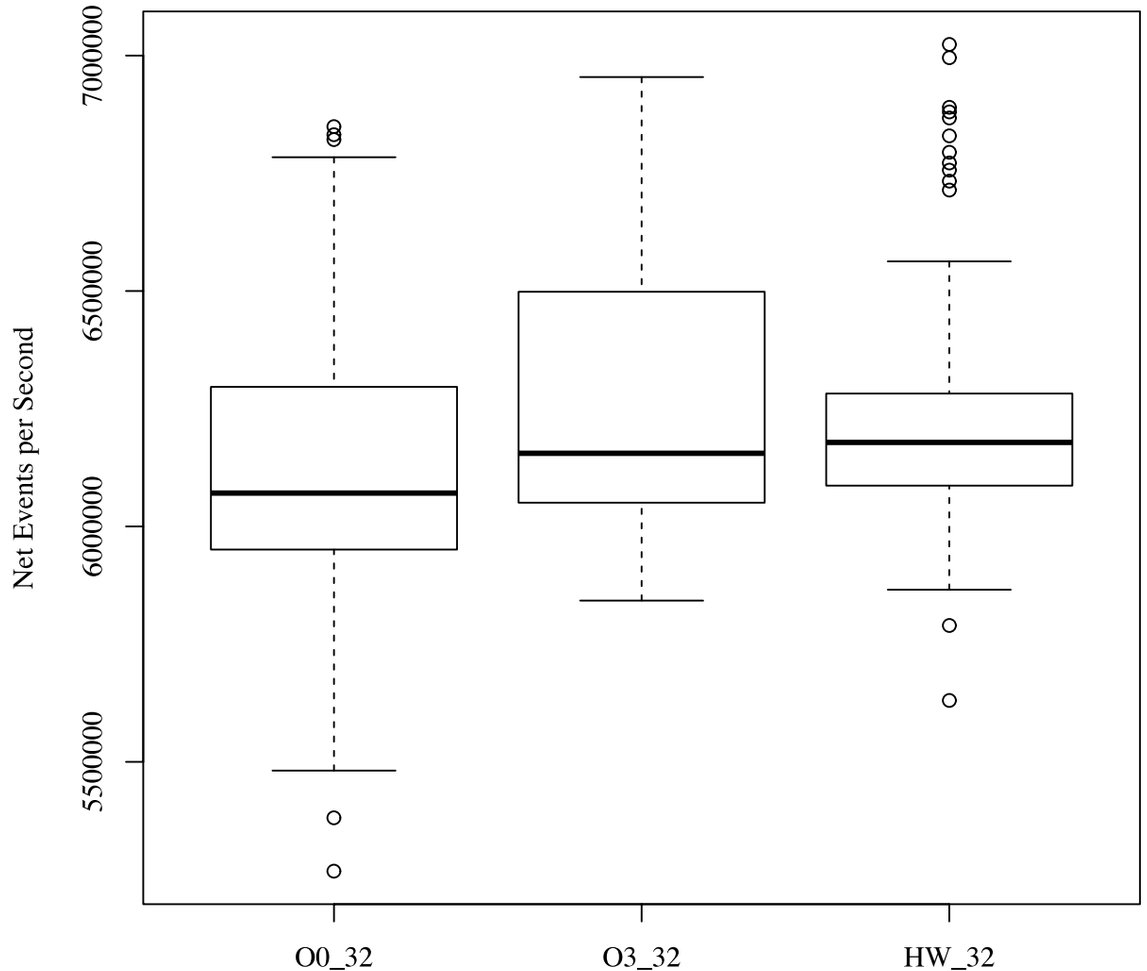


CFG for 'phold_event_handler' function

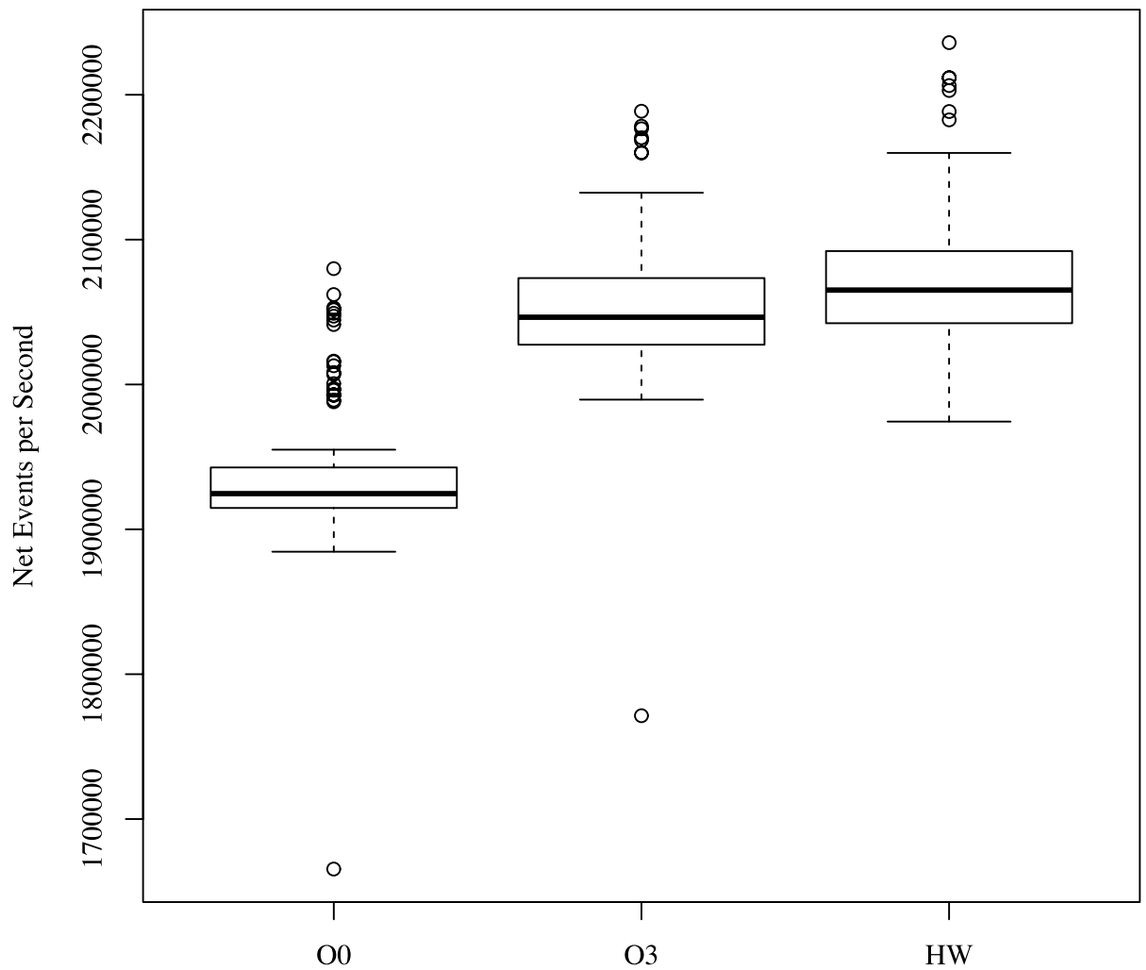
Instrumented!

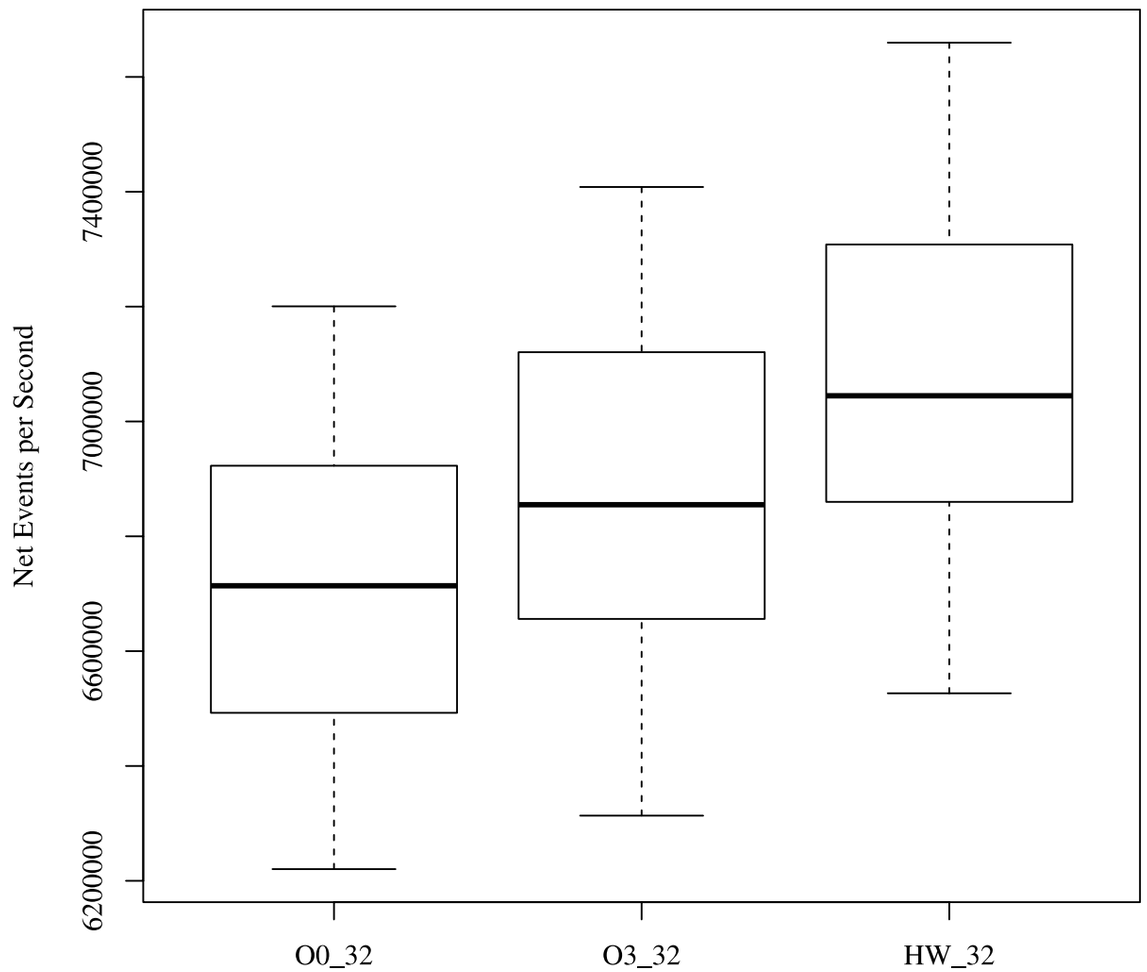
Results: phold model (2/32 cores)





Results: airport model (2/32 cores)





LPs are airports, events are planes traveling between airports, simple binary runway busy or free state

Related Work

- RCC (1999) - Perumalla / Fujimoto
  - Targeted C language
  - pragma based
- Backstroke / ROSE (2011/2012)- Vulov, Hou, Vuduc, Fujimoto, Jefferson, Quinlan
  - Targets C++
  - Source-to-source approach

Future Work

- Better loop support
- Inter-procedural analysis
  - Current approach requires a lot of inlining!
  - Need a general mechanism for tracking reversible vs un-reversible functions
- Bigger & better models!
- Dynamic memory allocation

Thank you!