# DAG Inlining: A Decision Procedure for Reachability-Modulo-Theories in Hierarchical Programs

*In: Programming Language Design and Implementation (PLDI) 2015*

Akash Lal and Shaz Qadeer

Microsoft Research

# Bug Finding via Bounded Verification

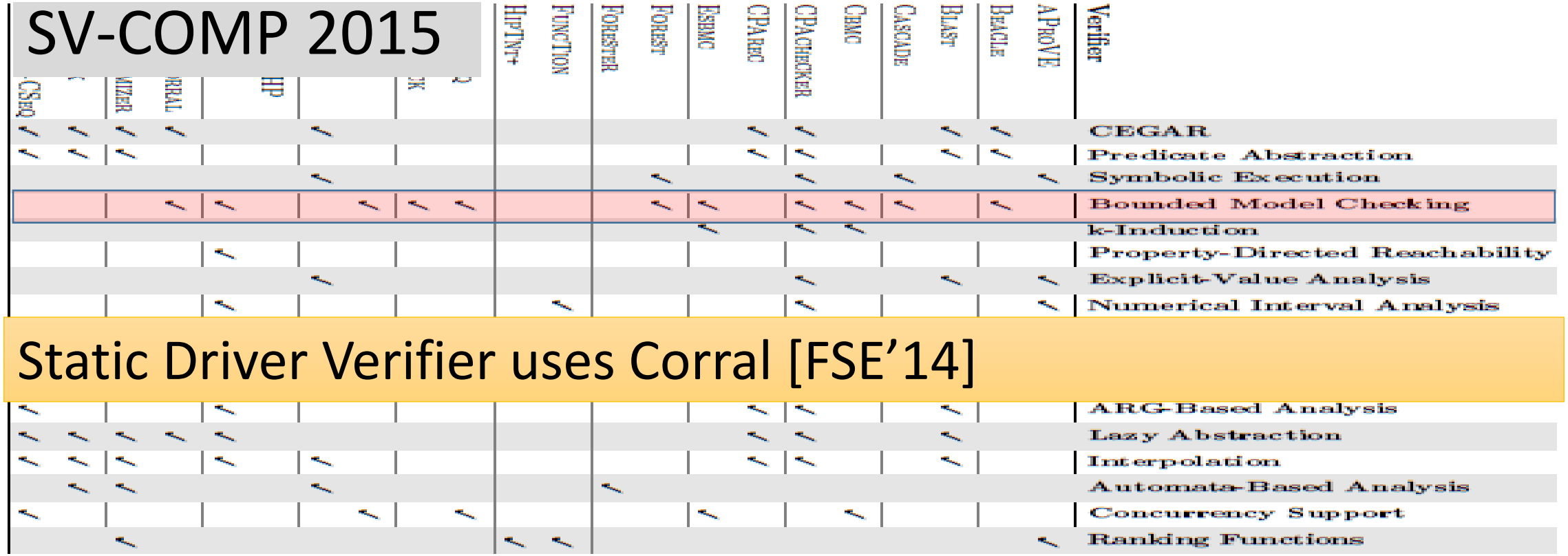
- Several success stories of automated verification
  - Static Driver Verifier, F-Soft, Facebook Infer, …
  - In finding bugs!

- Design for finding bugs quickly
  - Instead of discovering them as a by-product of proof failure

- Bounded verification: analyze a (useful) subset of program behaviors really fast

# Bounded Verification

- Build efficient decision procedures for Bounded Verification
  - Inspired by the success in Hardware verification



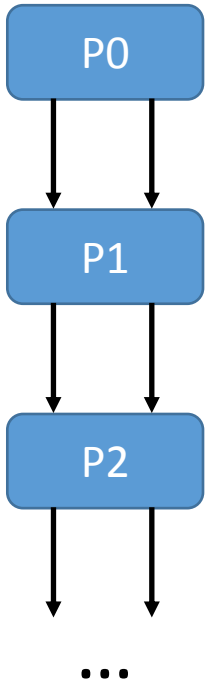
# Bounded Verification done previously


- Standard approach: Inline all procedures, generate a SAT/SMT constraint, invoke solver

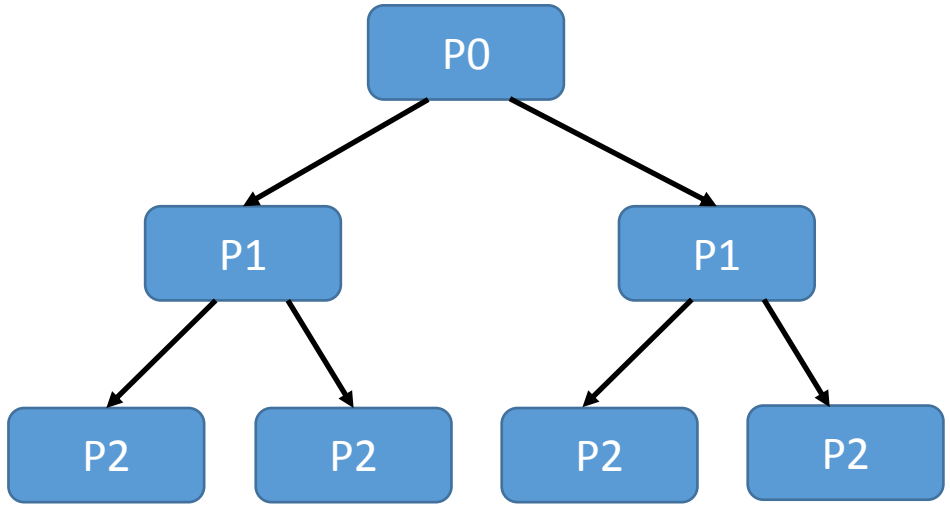
- Procedure Inlining causes exponential blowup

# Our Work: DAG Inlining

Program Call Graph



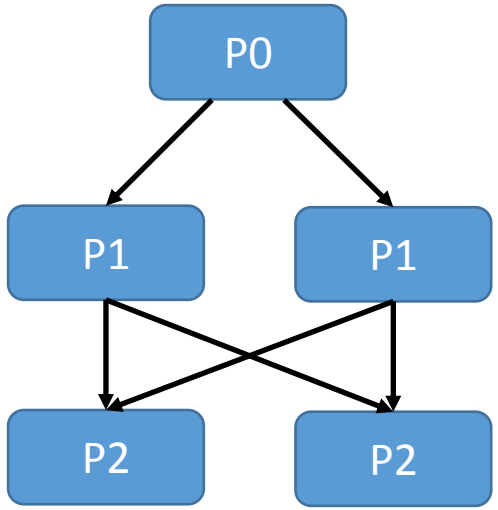

Standard Procedure Inlining




$2^i$ copies of $P_i$

Exponentially Blowup in Program Size

DAG Inlining

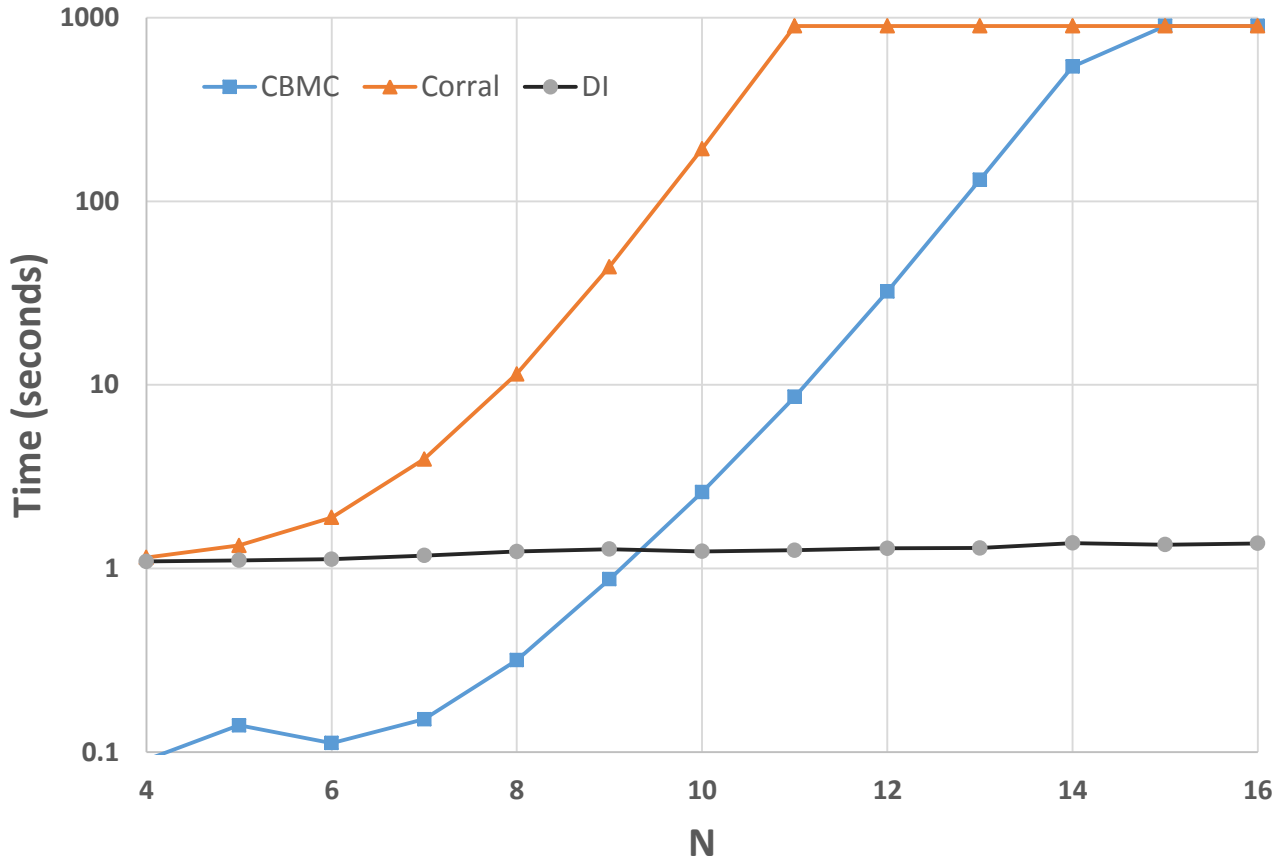



DAG structure program dependent

Potentially Exponential Savings!

# Micro Benchmark

```
var g: int;

procedure main() {
   g := 0;
   if(*) { call P0(); }
   else  { call P0(); }
}

procedure Pi() {
   g := g + 1;
   if(*) { call Pi+1(); }
   else  { call Pi+1(); }
}

procedure PN() {
   assert g == N;
}
```



# Talk Outline

- Problem definition
- VC Generation Algorithms [Background]
  - Single-procedure programs
  - Multi-procedure programs
- DAG Inlining Algorithm
- Evaluation

# Definitions

- Hierarchical Programs: Sequential programs without loops and recursion

- Reachability Modulo Theories (RMT): Find a terminating execution of a program whose operational semantics can be encoded in decidable logic.

# Programming Language

```
// variables
var x: T
// functions
func f: T → T

// commands
x := expr
assume expr
call foo(x);
```

SMT

```
// procedures
procedure foo(args) {
    LocalDecls;
    Body;
}

// control
if(expr) { cmds; } else { cmds; }
while(*) { cmds; }
```

Example:

- Linear arithmetic via the theory of linear arithmetic (LIA)
- Non-linear arithmetic via uninterpreted functions (EUF)
- Memory lookup using theory of arrays
- Floating-point as uninterpreted

# Reachability Modulo Theories

- Finding assertion failures is equivalent to finding terminating executions

```
assert e;
```

```
err := e;
if(!err) return;
```

```
call foo();
```

```
call foo();
if(!err) return;
```

```
main() {
   …
   return;
}
```

```
main() {
   err := true;
   …
   assume !err;
   return;
}
```

# Reachability Modulo Theories

- RMT in hierarchical programs is *decidable (NEXPTIME hard)* [RP'13]
  - Can't hide all sorts of complexity behind undecidability
- Direct application to bounded verification
  - E.g., "Bounded Model Checking"
- Relevant to unbounded verification
  - Checking inductive proofs (loop invariants) without pre-post conditions

# Talk Outline

- Problem definition
- VC Generation Algorithms ← Solving RMT  in Hierarchical programs
  - Single-procedure programs
  - Multi-procedure programs
- DAG Inlining Algorithm
- Evaluation

# VC Generation: Single Procedure

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto l1, l2;

  l1:
    x := x + 1;
    goto l3;

  l2:
    x := x + 2;
    goto l3;

  l3:
    assume !(x > y);
    return;
}
```

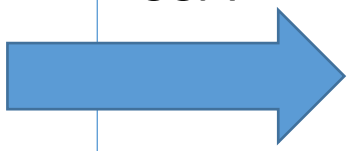
VC-Gen: f ➔ $\varphi_f(w, x, y)$

Theorem: $\varphi_f(w, x, y, z)$ holds iff f(w) can return (x,y)

Corollary: $\varphi_f$ is SAT iff f has a terminating execution

# VC Generation

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto l1, l2;

  l1:
    x := x + 1;
    goto l3;

  l2:
    x := x + 2;
    goto l3;

  l3:
    assume !(x > y);
    return;
}
```

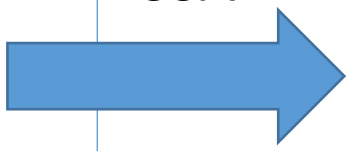



```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:
    x1 := *;
    y1 := x1 + w;
    goto l1, l2;

  l1:
    x2 := x1 + 1;
    goto l3;

  l2:
    x3 := x1 + 2;
    goto l3;

  l3:
    x4 := ϕ(x2, x3);
    assume !(x4 > y1);
    x := x4; y := y1;
    return;
}
```

# VC Generation

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto l1, l2;

  l1:
    x := x + 1;
    goto l3;

  l2:
    x := x + 2;
    goto l3;

  l3:
    assume !(x > y);
    return;
}
```

SSA



```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:
    x1 := *;
    y1 := x1 + w;
    goto l1, l2;

  l1:
    x2 := x1 + 1; x4 := x2;
    goto l3;

  l2:
    x3 := x1 + 2; x4 := x3;
    goto l3;

  l3:
    x4 := φ(x2, x3);
    assume !(x4 > y1);
    x := x4; y := y1;
    return;
}
```

# VC Generation

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:
    x := *;
    y := x + w;
    goto l1, l2;

  l1:
    x := x + 1;
    goto l3;

  l2:
    x := x + 2;
    goto l3;

  l3:
    assume !(x > y);
    return;
}
```

SSA+Passify

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:

    assume y1 == x1 + w;
    goto l1, l2;

  l1:
    assume x2 == x1 + 1; assume x4 == x2;
    goto l3;

  l2:
    assume x3 == x1 + 2; assume x4 == x3;
    goto l3;

  l3:

    assume !(x4 > y1);
    assume x == x4; assume y == y1;
    return;
}
```

# VC Generation

Block constraints

$C_{start}$: $y1 == x1 + w$

$C_{l1}$: $x2 == x1 + 1 \wedge x4 == x2$

$C_{l2}$: $x3 == x1 + 2 \wedge x4 == x3$

$C_{l3}$: $\neg(x4 > y1) \wedge x == x4 \wedge y == y1$

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:

    assume y1 == x1 + w;
    goto l1, l2;

  l1:
    assume x2 == x1 + 1; assume x4 == x2;
    goto l3;

  l2:
    assume x3 == x1 + 2; assume x4 == x3;
    goto l3;

  l3:

    assume !(x4 > y1);
    assume x == x4; assume y == y1;
    return;
}
```

# VC Generation

Algorithm

1. Introduce Boolean constant $b_L$ for each block L
2. $E_L$ is $b_L == C_L$ if L ends in return
   $b_L == C_L \wedge \vee_{m \in Succ(L)} b_m$
3. VC(f) is
   $b_{start} \wedge ( \wedge_{L \in Blocks(f)} E_L )$

Block constraints

$E_{start}$: $b_{start} == (y1 == x1 + w) \wedge (b_{l1} \vee b_{l2})$

$E_{l1}$: $b_{l1} == (x2 == x1 + 1 \wedge x4 == x2) \wedge b_{l3}$

$E_{l2}$: $b_{l2} == (x3 == x1 + 2 \wedge x4 == x3) \wedge b_{l3}$

$E_{l3}$: $b_{l3} == (\neg(x4 > y1) \wedge x == x4 \wedge y == y1)$

```
procedure f(w: int)
   returns (x: int, y: int)
{
  start:

    assume y1 == x1 + w;
    goto l1, l2;

  l1:
    assume x2 == x1 + 1; assume x4 == x2;
    goto l3;

  l2:
    assume x3 == x1 + 2; assume x4 == x3;
    goto l3;

  l3:

    assume !(x4 > y1);
    assume x == x4; assume y == y1;
    return;
}
```

# VC Generation: Multiple Procedures

```
procedure f(v1: int, v2: int)
   returns (r: int)
{
  var c: bool;
    goto l1, l2;

  l1:
    assume c;
    call r := g(v1);
    goto l3;

  l2:
    assume !c;
    call r := g(v2);
    goto l3;

  l3:
    return;
}
```

```
procedure g(a: int)
   returns (b: int)
{
   b := a + 1;
}
```

# VC Generation: Multiple Procedures

```
procedure f(v1: int, v2: int)
   returns (r: int)
{
  var c: bool;
    goto l1, l2;

  l1:
    assume c;
    call r := g(v1); assume M0;
    goto l3;

  l2:
    assume !c;
    call r := g(v2); assume M1;
    goto l3;

  l3:
    return;
}
```

```
procedure g(a: int)
   returns (b: int)
{
   b := a + 1;
}
```

VC(f): $(c \wedge M0) \vee (\neg c \wedge M1)$
VC(g): $b == a + 1$

Algorithm

1. Introduce Boolean constant $M_i$ for each call

# VC Generation: Multiple Procedures

```
procedure f(v1: int, v2: int)
   returns (r: int)
{
  var c: bool;
    goto l1, l2;

  l1:
    assume c;
    call r := g(v1); assume M0;
    goto l3;

  l2:
    assume !c;
    call r := g(v2); assume M1;
    goto l3;

  l3:
    return;
}
```

```
procedure g(a: int)
   returns (b: int)
{
   b := a + 1;
}
```

| | |
|---|---|
| $N_0$ | Execution starts in f |
| $\wedge\ N_0 \Rightarrow (c \wedge M_0) \vee (\neg c \wedge M_1)$ | VC of f |
| $\wedge\ M_0 \Rightarrow (N_1 \wedge v_1 == a_1 \wedge r == b_1)$ | formals equals actuals |
| $\wedge\ M_1 \Rightarrow (N_2 \wedge v_2 == a_2 \wedge r == b_2)$ | formals equals actuals |
| $\wedge\ N_1 \Rightarrow (b_1 == a_1 + 1)$ | VC of g |
| $\wedge\ N_2 \Rightarrow (b_2 == a_2 + 1)$ | VC of g |

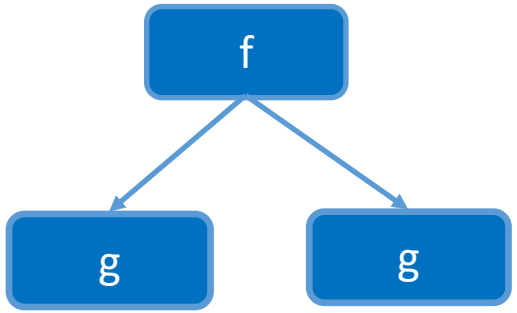
Algorithm

1. Introduce Boolean constant $M_i$ for each call
2. Introduce Boolean constant $N_i$ for each procedure instance
3. Connect

# VC Generation: Multiple Procedures

```
procedure f(v1: int, v2: int)
   returns (r: int)
{
  var c: bool;
    goto l1, l2;

  l1:
    assume c;
    call r := g(v1); assume M0;
    goto l3;

  l2:
    assume !c;
    call r := g(v2); assume M1;
    goto l3;

  l3:
    return;
}
```

```
procedure g(a: int)
   returns (b: int)
{
   b := a + 1;
}
```




Algorithm

1. Introduce Boolean constant $M_i$ for each call
2. Introduce Boolean constant $N_i$ for each procedure instance
3. Connect

# DAG Inlining: Algorithm

- *Dynamic Procedure Instances*: A procedure qualified by its call stack

```
procedure main()
{
   if(…) { A: bar(); }
   else  { B: baz(); }
}
```

```
procedure baz()
{
   C: foo();
}
```

```
procedure bar()
{
   D: foo();
}
```

`[A; C; foo]`

`[B; D; foo]`

- *Disjoint instances*: Two procedure instances that cannot be taken on the same execution

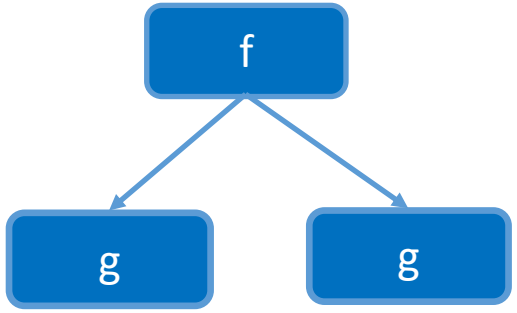
# DAG Inlining: Algorithm

- *Theorem*: Any two *disjoint* instances of the same procedure can be *merged* together when inlining.

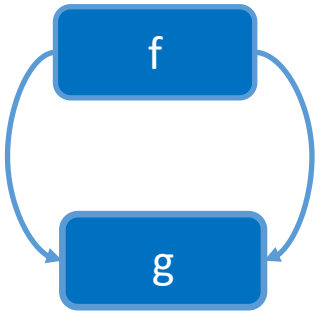
# DAG Inlining: Algorithm

```
procedure f(v1: int, v2: int)
   returns (r: int)
{
  var c: bool;
    goto l1, l2;

  l1:
    assume c;
    call r := g(v1);
    goto l3;

  l2:
    assume !c;
    call r := g(v2);
    goto l3;

  l3:
    return;
}
```

Standard (Tree) Inlining




DAG Inlining



# DAG Inlining: Algorithm

```
procedure f(v1: int, v2: int)
   returns (r: int)
{
  var c: bool;
    goto l1, l2;

  l1:
    assume c;
    call r := g(v1);
    goto l3;

  l2:
    assume !c;
    call r := g(v2);
    goto l3;

  l3:
    return;
}
```

Standard (Tree) Inlining

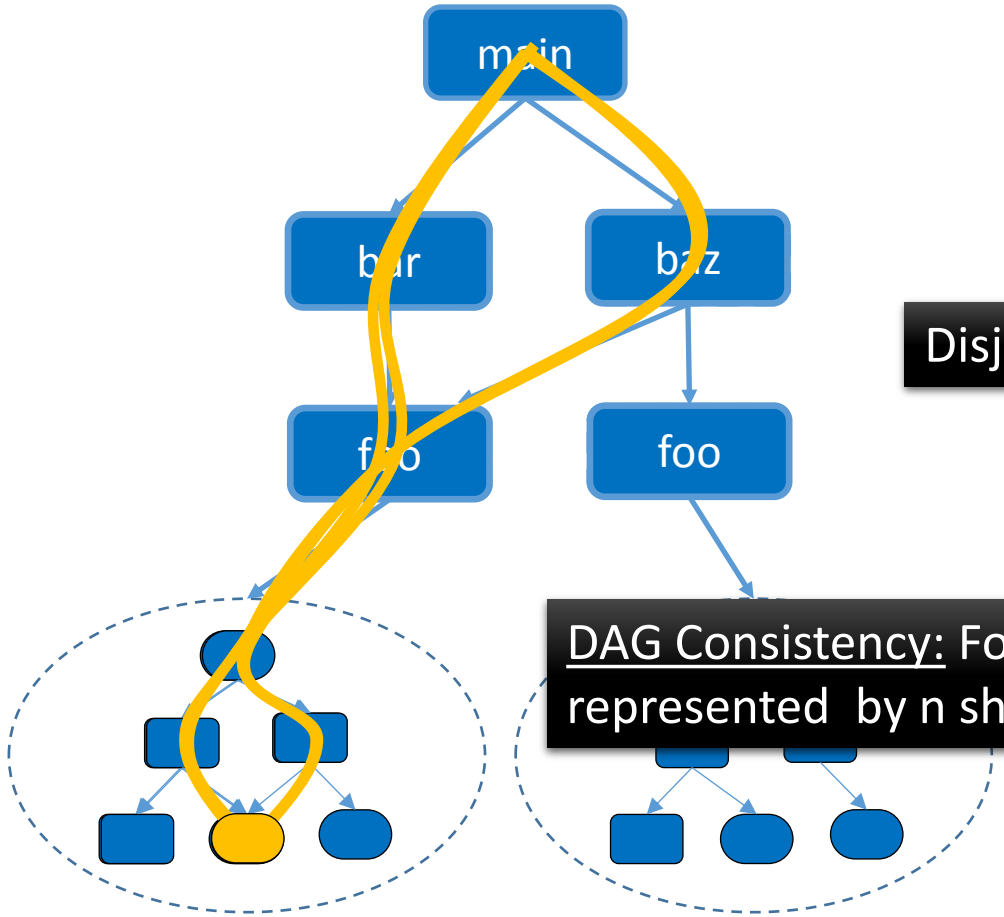
$N_0$ Execution starts in f
$\wedge N_0 \Rightarrow (c \wedge M_0) \vee (\neg c \wedge M_1)$ VC of f
$\wedge M_0 \Rightarrow (N_1 \wedge v_1 == a_1 \wedge r == b_1)$ formals equals actuals
$\wedge M_1 \Rightarrow (N_2 \wedge v_2 == a_2 \wedge r == b_2)$ formals equals actuals
$\wedge N_1 \Rightarrow (b_1 == a_1 + 1)$ VC of g
$\wedge N_2 \Rightarrow (b_2 == a_2 + 1)$ VC of g

DAG Inlining

$N_0$ Execution starts in f
$\wedge N_0 \Rightarrow (c \wedge M_0) \vee (\neg c \wedge M_1)$ VC of f
$\wedge M_0 \Rightarrow (N_1 \wedge v_1 == a_1 \wedge r == b_1)$ formals equals actuals
$\wedge M_1 \Rightarrow (N_1 \wedge v_2 == a_1 \wedge r == b_1)$ formals equals actuals
$\wedge N_1 \Rightarrow (b_1 == a_1 + 1)$ VC of g

$c \Rightarrow (r == v_1 + 1) \wedge \neg c \Rightarrow (r == v_2 + 1)$

# DAG Inlining: Algorithm



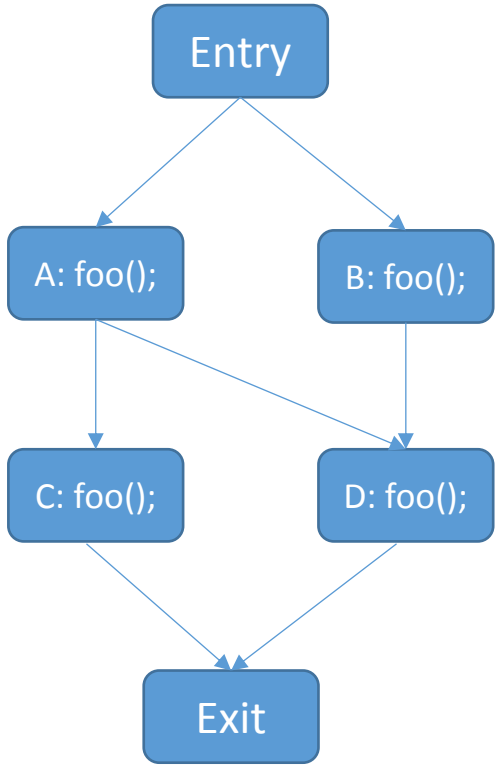

Disjoint([main; bar; foo], [main, baz, foo])

Disjoint(…)
Disjoint(…)

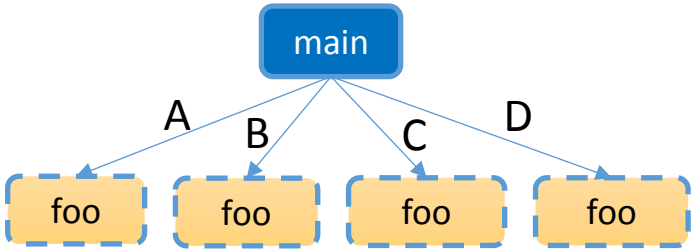
DAG Consistency: For each node n, all configurations represented by n should be mutually disjoint

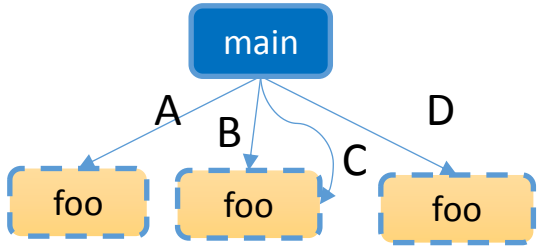
Algorithm: While inlining, keep merging with existing instances as long as the DAG is consistent

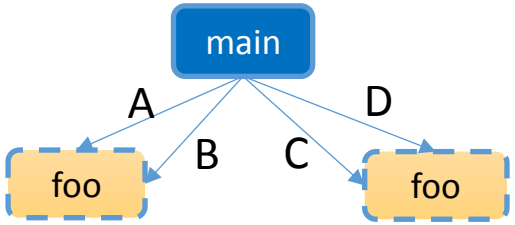
# Optimal merging reduces to Graph Coloring

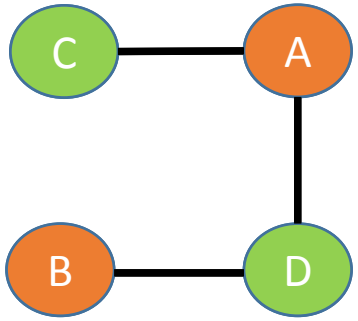



Control-Flow Graph

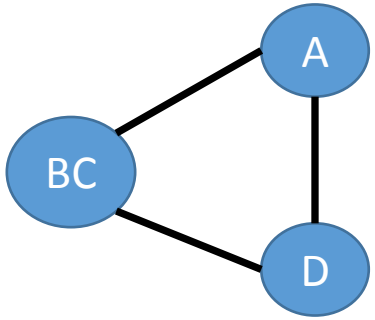



Conflict Graph:
Edge $\Rightarrow$ not disjoint

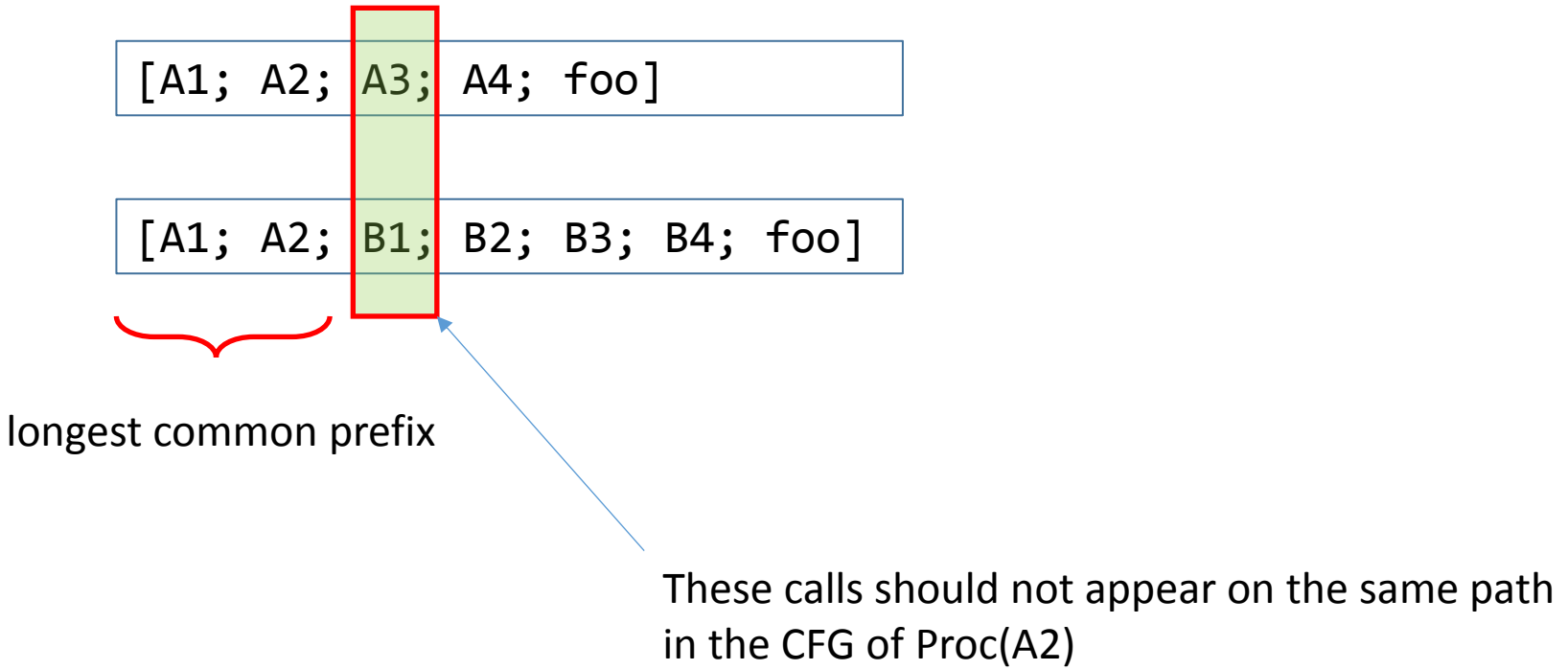




Optimal merging requires computation of maximal independent sets, which is equivalent to graph coloring

# Implementing the Algorithm

- We can decide disjointness in linear time based on control flow




- Disjointness of procedure instances can be resolved by disjointness in a single CFG.

# Implementing the Algorithm [See Paper]

- Disjointness of two configurations in linear time
- Deciding DAG consistency in quadratic time

- Greedy graph coloring (8% off the optimal)

- Overall: Less than 0.4% time spent in DAG operations

# Experiments: Static Driver Verifier

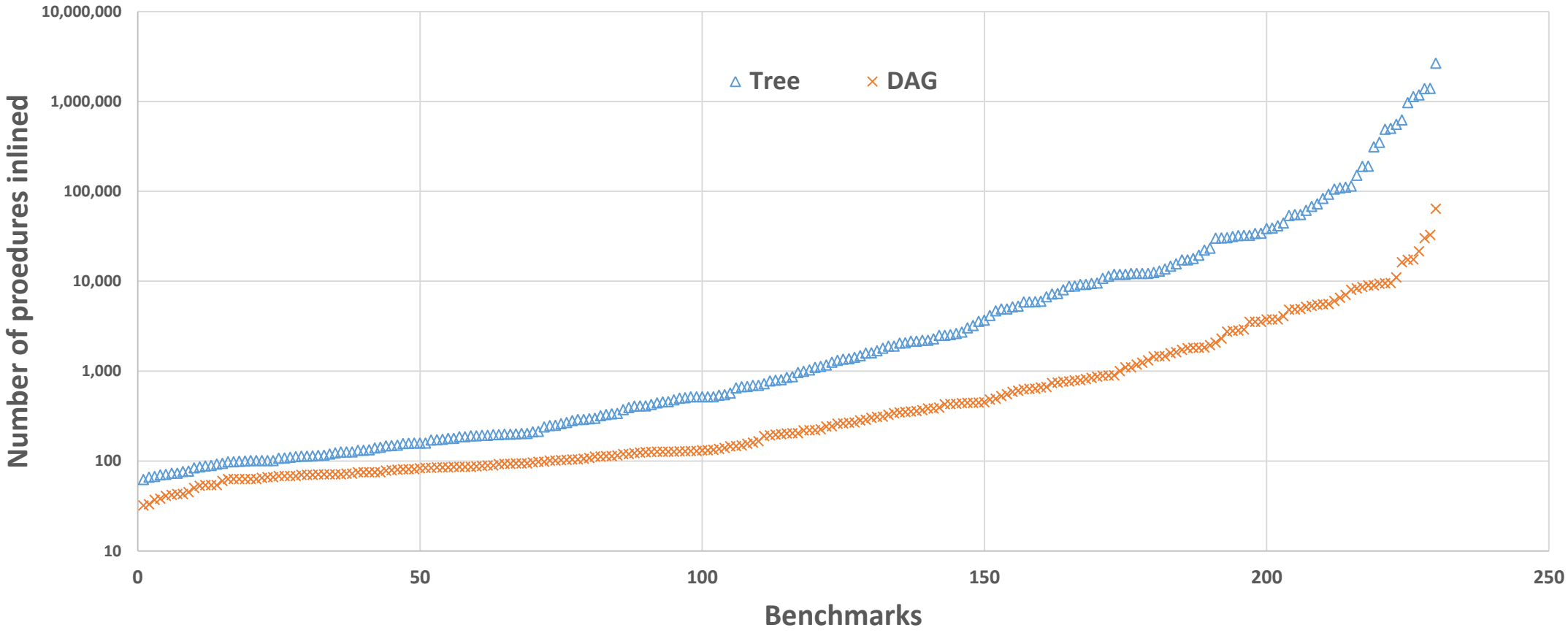
- Commercial tool, ships with Windows
  - Used internally and by third-party driver developers
  - Part of Windows Driver Certification program

- Uses Corral, an RMT solver
  - Based on Tree Inlining, but:
  - Includes several optimizations over tree inlining

- Total LOC: 800K
- Total verification time: well over a month

# Compression in Practice

- Tree/DAG Sizes



# Compression strategy

- We use a greedy merging strategy: turns out to be around 8% off the optimal in practice

| Tree | OPT | FIRST | MAXC | RANDOM | RANDOMPICK |
|---|---|---|---|---|---|
| 312231 | 1582 | 1673 | 1629 | 5096 | 1760 |
| 348329 | 16223 | 16261 | 16263 | 18539 | 16256 |
| 486713 | 9379 | 9751 | 9960 | 15406 | 10376 |
| 499799 | 4854 | 6400 | 6920 | T/O | 10191 |
| 553790 | 2793 | 2846 | 2848 | 8555 | 2860 |
| 621285 | 17182 | 22589 | 22545 | T/O | 24921 |
| 964394 | 1796 | 1890 | 1878 | 5258 | 2231 |
| 1125448 | 21459 | 22068 | 21997 | 42743 | 22733 |
| 1177613 | 17419 | 17576 | 17557 | 30873 | 18463 |
| 1384747 | 8315 | 8527 | 8484 | T/O | 10268 |
| 1390941 | 4887 | 4999 | 5109 | 13327 | 5306 |
| 2645020 | 8623 | 8798 | 9017 | 18246 | 9270 |
| 3211353 | T/O | 13782 | 13837 | 18976 | 13847 |
| Dev: | – | 8% | 9% | 129% | 21% |

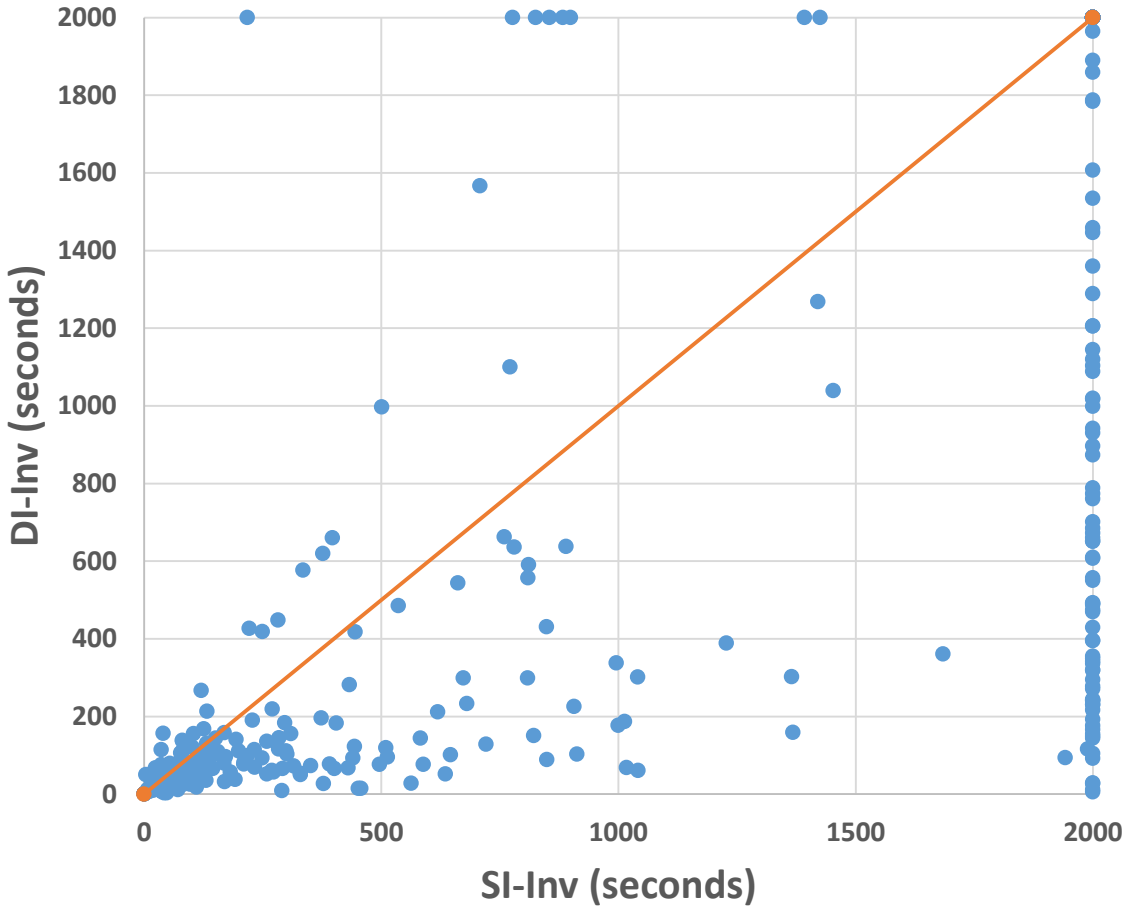
# Results: Summary

| Algorithm | #TO | #Bugs | #Inlined (avg.) | Time (1000 s) Bug | Time (1000 s) No-bug |
|---|---|---|---|---|---|
| Tree-1 | 418 | 51 | 885.2 | 9.1 | 50.7 |
| DAG-1 | 354 | 64 | 271.6 | 5.4 | 25.2 |
| Tree-2 | 358 | 72 | 759.3 | 13.6 | 82.6 |
| DAG-2 | 280 | 83 | 272.4 | 9.3 | 44.3 |

Production Quality

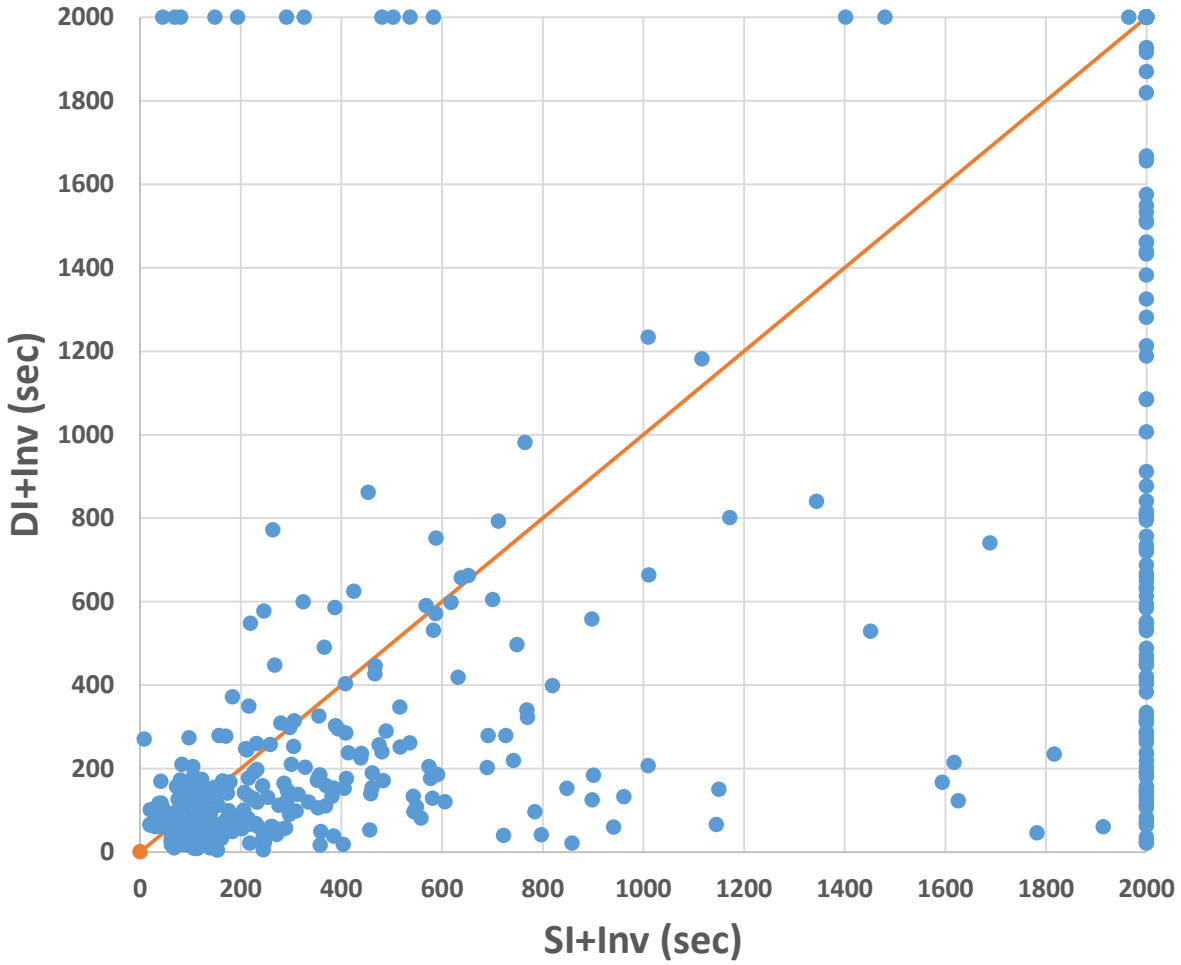
- Find more bugs in less time
- Almost twice as fast as the production system

# Results - 1




- Number of instances: 619
- Reduction in Timeouts: 64
- 5X speedup: 35
- 5X slowdown: 2

# Results - 2




- Number of instances: 619
- Reduction in Timeouts: 78
- 5X speedup: 45
- 5X slowdown: 1

# Before the last slide …

# Microsoft Research India

We are hiring!

Researchers, post-docs & engineers

Systems, ML, Crypto, Theory, PL, HCI, ICT4D,…

# Summary

- Reachability in Hierarchical Programs is fundamental
- Standard (Tree) inlining causes exponential blowup
  - Limits many BMC tools to small programs
- DAG inlining refines the age-old idea of procedure inlining
  - Demonstrated significant speedups of a production system