# Compositional Bitvector Analysis For Concurrent Programs With Nested Locks

Zachary Kincaid
joint work with Azadeh Farzan

University of Toronto

September 14, 2010

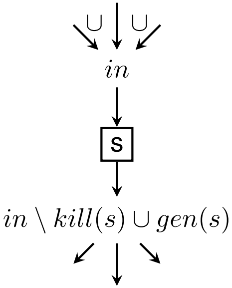
# Contribution

Algorithm for solving bitvector problems for concurrent programs

- Handles dynamic synchronization *precisely*
- Thread compositional
  - Scales in # of threads
- Solves the problem for every fact and every location simultaneously

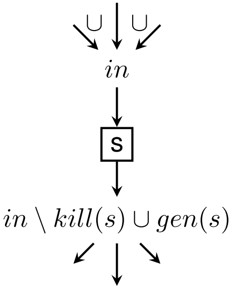
# Bitvector Analyses

- Let $D$ be a finite set of data flow facts
- For each statement $s$, define
  - $gen(s) \subseteq D$: set of facts generated by $s$
  - $kill(s) \subseteq D$: set of facts killed by $s$
  - $[\![s]\!](in) = (in \setminus kill(s)) \cup gen(s)$



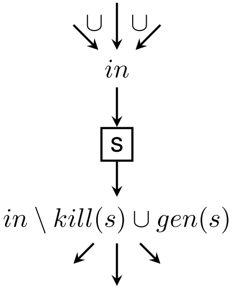
# Bitvector Analyses

- Let $D$ be a finite set of data flow facts
- For each statement $s$, define
  - $gen(s) \subseteq D$: set of facts generated by $s$
  - $kill(s) \subseteq D$: set of facts killed by $s$
  - $[\![s]\!](in) = (in \setminus kill(s)) \cup gen(s)$
- Let $D = \mathbb{B}^n$ ($\mathbb{B} = \{\mathit{tt}, \mathit{ff}\}$)
- $[\![s]\!](\langle in_1, \ldots, in_n \rangle) = \langle [\![s]\!]_1(in_1), \ldots, [\![s]\!]_n(in_n) \rangle$



# Bitvector Analyses

- Let $D$ be a finite set of data flow facts
- For each statement $s$, define
  - $gen(s) \subseteq D$: set of facts generated by $s$
  - $kill(s) \subseteq D$: set of facts killed by $s$
  - $[\![s]\!](in) = (in \setminus kill(s)) \cup gen(s)$
- Let $D = \mathbb{B}^n$ ($\mathbb{B} = \{tt, ff\}$)
- $[\![s]\!](\langle in_1, \ldots, in_n \rangle) = \langle [\![s]\!]_1(in_1), \ldots, [\![s]\!]_n(in_n) \rangle$




- Examples: Reaching definitions, available expressions, live variables, ...
- We will concentrate on *forwards flow, may analyses* (e.g. reaching definitions)

# Related work

- Parallelism for free! [Knoop et al, TOPLAS96]
  - Precise bitvector analysis for cobegin/coend parallelism
  - Some generalizations [Esparza & Knoop, FOSSACS99; Esparza & Podelski, POPL00; Seidl & Steffen, ESOP00; Knoop, Euro-Par98]
- Nested locks [Kahlon & Gupta, POPL07]
  Determine whether two local paths (run suffixes) can be interleaved
  - Compute local lock information for each path
    - Locksets, acquisition histories
  - Consistency check on local lock information

# Program model

- Finite set of threads
  - Optional: infinitely many copies of each thread run simultaneously
- Finite set of locks
- All threads start executing at the beginning of the program
- No locks are held in the initial state
- Each thread releases locks in the reverse order they were acquired

Not allowed: `acq(l); acq(m); rel(l); rel(m)`

# Concurrent Bitvector Analyses

- Optimal solution (sequential case): meet over paths
- Optimal solution (concurrent case): meet over feasible runs
  - A run is *feasible if*
    - When projected onto a single thread, it corresponds to a path in the CFG
    - No two threads hold the same lock simultaneously

# Concurrent Bitvector Analyses

- Optimal solution (sequential case): meet over paths
- Optimal solution (concurrent case): meet over feasible runs
  - A run is *feasible if*
    - When projected onto a single thread, it corresponds to a path in the CFG
    - No two threads hold the same lock simultaneously

| Thread 1 | Thread 2 |
| --- | --- |
| `acquire(l)` | `acquire(m)` |
| `acquire(m)` | `x = 0` |
| `x = x + 1` | `x = 1` |
| `release(m)` | `y = 0` |
| `x = x * 2` | `release(m)` |
| `release(l)` | |

# Concurrent Bitvector Analyses

- Optimal solution (sequential case): meet over paths
- Optimal solution (concurrent case): meet over feasible runs
  - A run is *feasible if*
    - When projected onto a single thread, it corresponds to a path in the CFG
    - No two threads hold the same lock simultaneously

| Thread 1 | Thread 2 | Feasible run | Infeasible run |
|---|---|---|---|
| `acquire(l)` | `acquire(m)` | `acquire(l)` | `acquire(m)` |
| `acquire(m)` | `x = 0` | `acquire(m)` | `acquire(l)` |
| `x = x + 1` | `x = 1` | `x = x + 1` | `x = 0` |
| `release(m)` | `y = 0` | `release(m)` | `acquire(m)` |
| `x = x * 2` | `release(m)` | `acquire(m)` | `x = x + 1` |
| `release(l)` | | `x = 0` | |
| | | `x = x * 2` | |

Threads 1 & 2 hold `m`

# Motivation

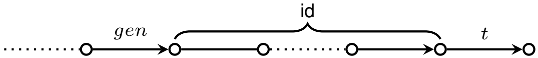
- Optimization
- Reaching definitions analysis can be used to construct *dependence graphs*, which may be useful for:
  - Slicing
  - Bootstrapping more sophisticated analyses (e.g. interval analysis)

# Parallelism for free!

Observation: there are 3 monotone functions on $\mathbb{B}$:

- $id = \lambda x.x$
- $gen = \lambda x.\mathit{tt}$
- $kill = \lambda x.\mathit{ff}$

A fact $f$ *reaches* $t$ iff there exists a witness run:

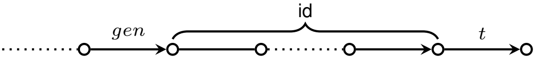


# Parallelism for free!

Observation: there are 3 monotone functions on $\mathbb{B}$:

- $id = \lambda x.x$ `y = 0`
- $gen = \lambda x.tt$ `x = 1`
- $kill = \lambda x.ff$ `x = 0`

A fact $f$ *reaches* $t$ iff there exists a witness run:




Witness

```
acquire(m)
z = 2
acquire(l)
release(m)
acquire(m)
x = 0
x = 1
y = 0
release(m)
z = z - 1
x = x + 1
```

# Projection

- For every feasible run $\rho$ and every thread $T$, there exists a second feasible run with the same transitions in the same order, except no transitions of $T$ are executed [Kahlon et al., CAV05]

Witness

```
acquire(m)
z = 2
acquire(l)
release(m)
acquire(m)
x = 0
x = 1
y = 0
release(m)
z = z - 1
x = x + 1
```

# Projection

- For every feasible run $\rho$ and every thread $T$, there exists a second feasible run with the same transitions in the same order, except no transitions of $T$ are executed [Kahlon et al., CAV05]
- For every *witness* $\rho$ and every thread $T$, there exists a second *witness* with the same transitions in the same order, except no transitions of $T$ are executed

Witness

```
acquire(m)
z = 2
acquire(l)
release(m)
acquire(m)
x = 0
x = 1
y = 0
release(m)
z = z - 1
x = x + 1
```

# Projection

- For every feasible run $\rho$ and every thread $T$, there exists a second feasible run with the same transitions in the same order, except no transitions of $T$ are executed [Kahlon et al., CAV05]
- For every *witness* $\rho$ and every thread $T$, there exists a second *witness* with the same transitions in the same order, except no transitions of $T$ are executed

Witness

```
acquire(l)
acquire(m)
x = 0
x = 1
y = 0
release(m)
x = x + 1
```

# Projection

- For every feasible run $\rho$ and every thread $T$, there exists a second feasible run with the same transitions in the same order, except no transitions of $T$ are executed [Kahlon et al., CAV05]
- For every *witness* $\rho$ and every thread $T$, there exists a second *witness* with the same transitions in the same order, except no transitions of $T$ are executed
- If there is a witness for a $t$, there is a witness involving 1 or 2 threads

Witness

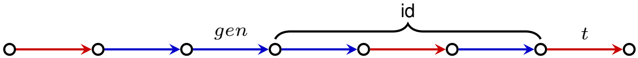
```
acquire(l)
acquire(m)
x = 0
x = 1
y = 0
release(m)
x = x + 1
```

# Strategy

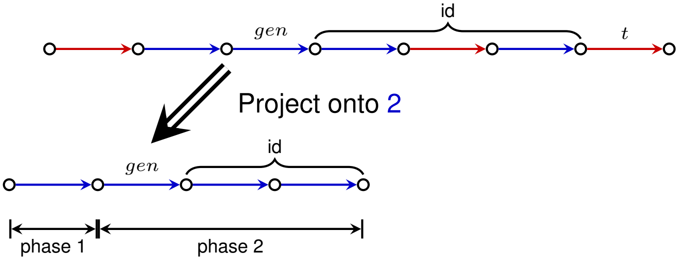
Compute 1-thread and 2-thread witnesses, then combine the results

- 1-thread witness computation = sequential bitvector analysis
- 2-thread witness computation
  - For each thread, compute its set of generating and preserving runs
  - For each pair of transitions from different threads, determine whether there is a generating run and preserving run that can be interleaved
- Questions:
  1. When can generating and preserving run can be interleaved?
  2. How can the (possibly infinite) sets of generating and preserving runs be represented?

# Local structure of witnesses

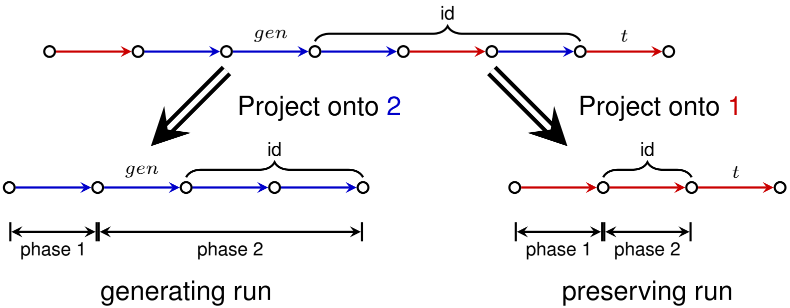


# Local structure of witnesses




generating run

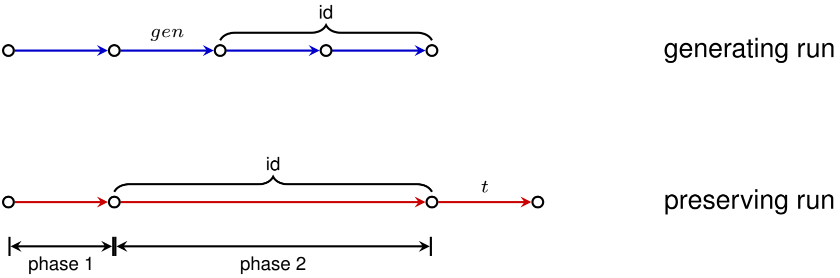
# Local structure of witnesses



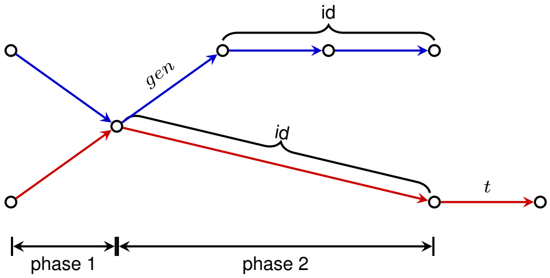
# Local structure of witnesses

| Generating run | Witness | Preserving run |
| --- | --- | --- |
| `acquire(m)` | `acquire(l)` | `acquire(l)` |
| `x = 0` | `acquire(m)` | `acquire(m)` |
| `x = 1` | `x = 0` | `x = x + 1` |
| `y = 0` | `x = 1` | |
| `release(m)` | `y = 0` | |
| | `release(m)` | |
| | `acquire(m)` | |
| | `x = x + 1` | |

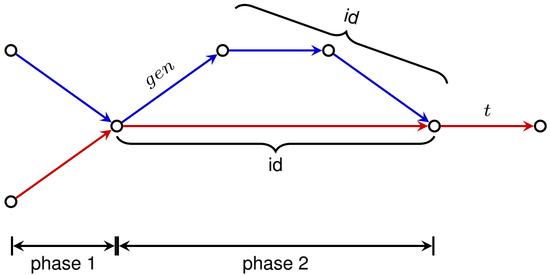
# Compositional approach to bitvector analysis



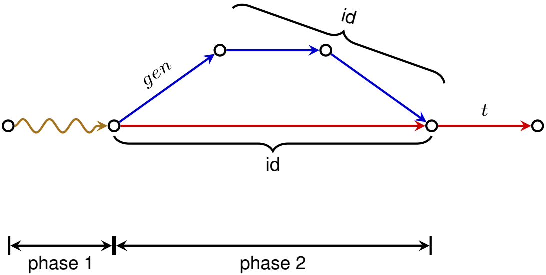
# Compositional approach to bitvector analysis



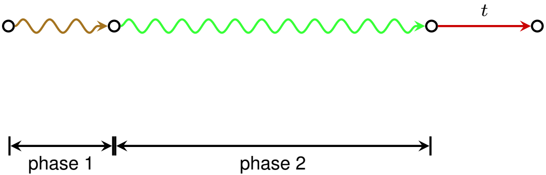
# Compositional approach to bitvector analysis



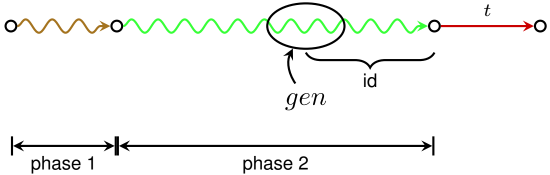
# Compositional approach to bitvector analysis



# Compositional approach to bitvector analysis



# Compositional approach to bitvector analysis



# Compositional approach to bitvector analysis

Generating run

```
acquire(m)
x = 0
x = 1
y = 0
release(m)
```

Preserving run

```
acquire(l)
acquire(m)
x = x + 1
```

# Compositional approach to bitvector analysis

Generating run

```
acquire(m)
x = 0
```

```
x = 1
y = 0
release(m)
```

Witness

```
acquire(l)
acquire(m)
x = 0
```

```
x = 1
y = 0
release(m)
acquire(m)
```

```
x = x + 1
```

Witness

```
acquire(m)
x = 0
acquire(l)
```

```
acquire(m)
x = 1
y = 0
release(m)
```

```
x = x + 1
```

Preserving run

```
acquire(l)
```

```
acquire(m)
```

```
x = x + 1
```

# Strategy

Compute 1-thread and 2-thread witnesses, then combine the results

- 1-thread witness computation = sequential bitvector analysis
- 2-thread witness computation
  - For each thread, compute its set of generating and preserving runs
  - For each pair of transitions from different threads, determine whether there is a generating run and preserving run that can be interleaved
- Questions:
  1. When can generating and preserving run can be interleaved?
  2. How can the (possibly infinite) sets of generating and preserving runs be represented?

# Collecting generating and preserving runs

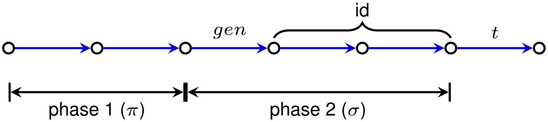
- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$

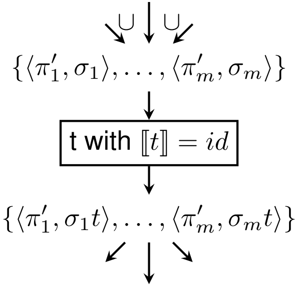
# Collecting generating and preserving runs

- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
  - Domain: Set of pairs of paths
  - Input: Set of generating runs to $t$
  - Output: Set of generating runs through $t$

# Collecting generating and preserving runs

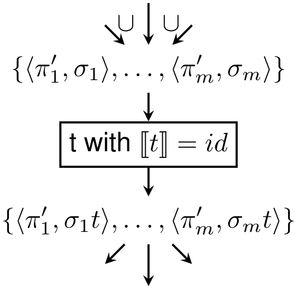
- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
  - Domain: Set of pairs of paths
  - Input: Set of generating runs to $t$
  - Output: Set of generating runs through $t$

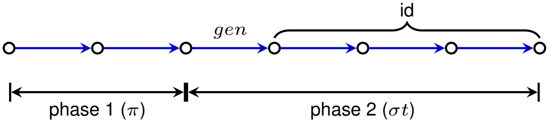


# Collecting generating and preserving runs

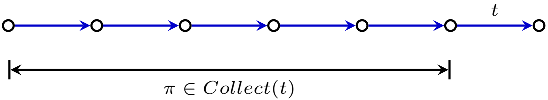
- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
  - Domain: Set of pairs of paths
  - Input: Set of generating runs to $t$
  - Output: Set of generating runs through $t$

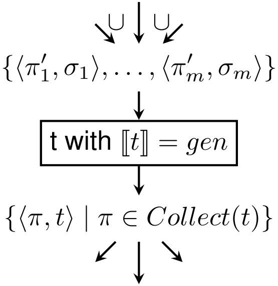


# Collecting generating and preserving runs

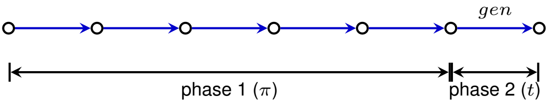
- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
  - Domain: Set of pairs of paths
  - Input: Set of generating runs to $t$
  - Output: Set of generating runs through $t$

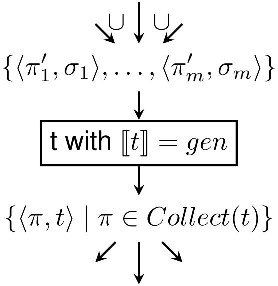


# Collecting generating and preserving runs

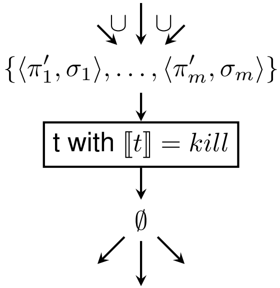
- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
  - Domain: Set of pairs of paths
  - Input: Set of generating runs to $t$
  - Output: Set of generating runs through $t$



# Collecting generating and preserving runs

- Collecting trace semantics for $t$:
    - Domain: Set of traces
    - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
    - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
    - Domain: Set of pairs of paths
    - Input: Set of generating runs to $t$
    - Output: Set of generating runs through $t$



# Collecting generating and preserving runs

- Collecting trace semantics for $t$:
  - Domain: Set of traces
  - Input: Set of traces to $t$ - $\{\pi_1, \cdots, \pi_n\}$
  - Output: Set of traces through $t$ - $\{\pi_1 t, \cdots, \pi_n t\}$
- LFP solution = $Collect(t)$: set of all traces to $t$
- Generating runs:
  - Domain: Set of pairs of paths
  - Input: Set of generating runs to $t$
  - Output: Set of generating runs through $t$
- LFP solution = $Generating(t)$: set of all generating runs to $t$
- Preserving runs can be computed similarly

# Strategy

Compute 1-thread and 2-thread witnesses, then combine the results

- 1-thread witness computation = sequential bitvector analysis
- 2-thread witness computation
  - For each thread, compute its set of generating and preserving runs
  - For each pair of transitions from different threads, determine whether there is a generating run and preserving run that can be interleaved
- Questions:
  1. When can generating and preserving run can be interleaved?
  2. How can the (possibly infinite) sets of generating and preserving runs be represented?

# Related work

- Parallelism for free! [Knoop et al, TOPLAS96]
  - Precise bitvector analysis for cobegin/coend parallelism
  - Some generalizations [Esparza & Knoop, FOSSACS99; Esparza & Podelski, POPL00; Seidl & Steffen, ESOP00; Knoop, Euro-Par98]
- Nested locks [Kahlon & Gupta, POPL07]
  Determine whether two local paths (run suffixes) can be interleaved
  - Compute local lock information for each path
    - Locksets, acquisition histories
  - Consistency check on local lock information

# Related work

- Parallelism for free! [Knoop et al, TOPLAS96]
  - Precise bitvector analysis for cobegin/coend parallelism
  - Some generalizations [Esparza & Knoop, FOSSACS99; Esparza & Podelski, POPL00; Seidl & Steffen, ESOP00; Knoop, Euro-Par98]
- Nested locks [Kahlon & Gupta, POPL07]
  Determine whether two local paths (run suffixes) can be interleaved
  - Compute local lock information for each path
    - Locksets, acquisition histories
  - Consistency check on local lock information

# Computing generating & preserving runs

- $\mathcal{L}$: local lock information from Kahlon & Gupta
  - Abstraction function: compute local lock information component-wise:

$$\alpha(\{\pi_1, \pi_2, \cdots\}) = \{info(\pi_1), info(\pi_2), ...\}$$

# Computing generating & preserving runs

- $\mathcal{L}$: local lock information from Kahlon & Gupta
  - Abstraction function: compute local lock information component-wise:

$$\alpha(\{\pi_1, \pi_2, \cdots\}) = \{info(\pi_1), info(\pi_2), ...\}$$

- Domain for computing $Collect(t)$: $\langle \wp(\mathcal{L}); \subseteq \rangle$

# Computing generating & preserving runs

- $\mathcal{L}$: local lock information from Kahlon & Gupta
  - Abstraction function: compute local lock information component-wise:

$$\alpha(\{\pi_1, \pi_2, \cdots\}) = \{info(\pi_1), info(\pi_2), ...\}$$

- Domain for computing $Collect(t)$: $\langle \wp(\mathcal{L}); \subseteq \rangle$
- Domain for computing $Generating/Preserving(t)$: $\langle \wp(\underbrace{\mathcal{L}}_{\text{phase 1}} \times \underbrace{\mathcal{L}}_{\text{phase 2}}); \subseteq \rangle$

# One fact to many

- Domain for generating runs for a single fact: $\langle \wp(\mathcal{L} \times \mathcal{L}); \subseteq \rangle$

# One fact to many

- Domain for generating runs for a single fact: $\langle \mathcal{L} \times \mathcal{L} \to \mathbb{B}; \preceq \rangle$

# One fact to many

- Domain for generating runs for a single fact: $\langle \mathcal{L} \times \mathcal{L} \to \mathbb{B}; \preceq \rangle$
- Domain for generating runs for all facts: $\langle \mathcal{L} \times \mathcal{L} \to \mathbb{B}^n; \preceq \rangle$

# One fact to many

- Domain for generating runs for a single fact: $\langle \mathcal{L} \times \mathcal{L} \to \mathbb{B}; \preceq \rangle$
- Domain for generating runs for all facts: $\langle \mathcal{L} \times \mathcal{L} \to \mathbb{B}^n; \preceq \rangle$
- Represent a function as a subset of $(\mathcal{L} \times \mathcal{L}) \times \mathbb{B}^n$; if $r \in \mathcal{L} \times \mathcal{L}$ doesn't appear in the representation, it is associated with $\mathit{ff}^n$.

# The algorithm

1. Sequential data flow analysis

# The algorithm

1. Sequential data flow analysis
2. Compute generating runs for each thread

# The algorithm

1. Sequential data flow analysis
2. Compute generating runs for each thread
3. Build summaries
   - Throw away “end” transitions for generating runs
   - Join over all transitions ⇒ *Thread summary*
   - Join over all threads ⇒ *Program summary* (parameterized systems)

# The algorithm

1. Sequential data flow analysis
2. Compute generating runs for each thread
3. Build summaries
   - Throw away "end" transitions for generating runs
   - Join over all transitions $\Rightarrow$ *Thread summary*
   - Join over all threads $\Rightarrow$ *Program summary* (parameterized systems)
4. For each thread $T$ in the program:

   For each transition $t$ in $T$:

   For each preserving run $\langle r, D \rangle$ reaching $t$:

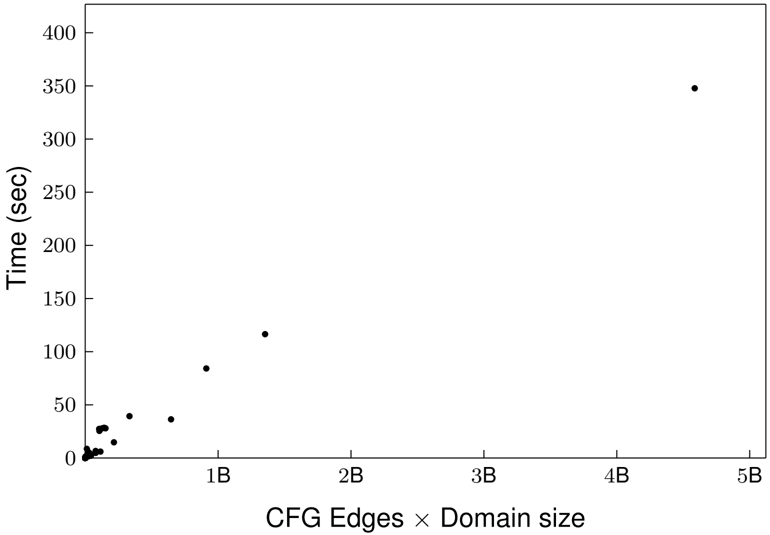
   For each generating run $\langle r', D' \rangle$ in the summary:

   If $r$ and $r'$ can be interleaved, $D \cap D'$ reaches $t$.

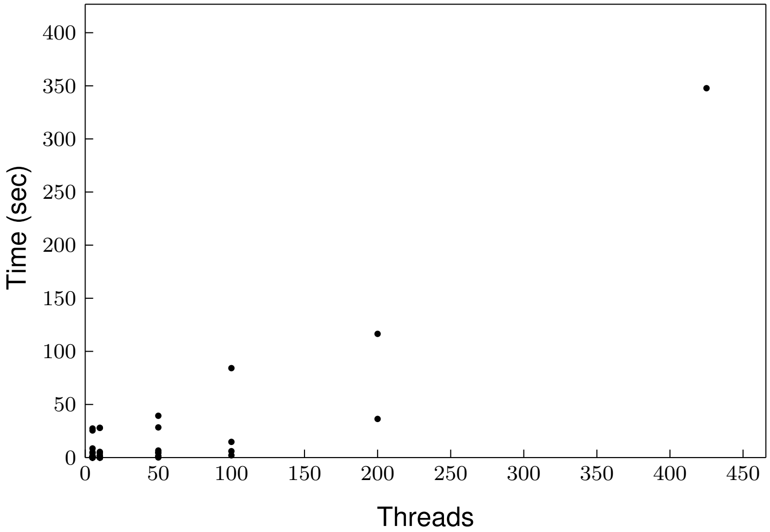
# Experiments

- Implementation applies to C with pthreads
- Ran reaching definitions on FUSE
  - Inlined all functions
  - (Unsound) alias analysis to finitize set of locks
  - Split FUSE into chunks of 5, 10, 50, 100, 200, 425 functions
  - Each function is considered a different thread

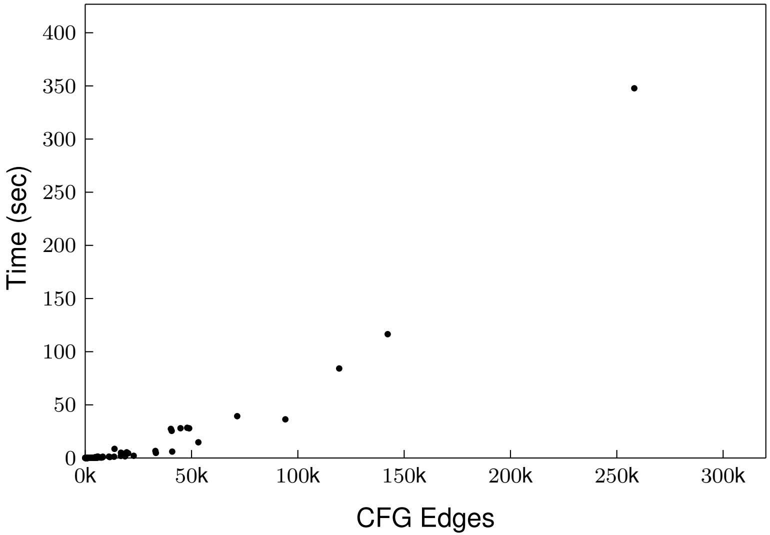
# Experiments



# Experiments



# Experiments



# Overview

Algorithm for computing the *optimal* solution to bitvector problems for concurrent programs communicating via nested locks

- Compositional: scales in # of threads
- Scales in # of data flow facts

# Questions?

Thank you for your attention.