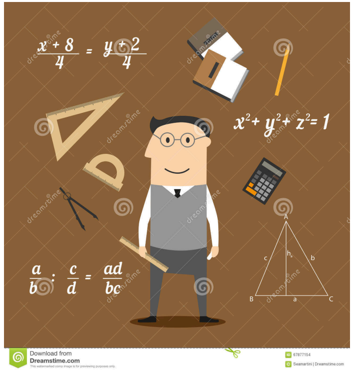
"Systemized" Static Analysis

Harry Xu

University of California, Los Angeles

Overview of My Work



Systems

PL

Static Analysis: Has the Problem Been Solved?

Academia

- Hundreds of papers published in the past decade
- Algorithms become increasingly sophisticated

More elegant

Industry

- Less than a dozen commercial analysis tools
- Use very simple algorithms
- Software becomes increasingly large and dynamic

More practical

The Ever-increasing Gap

Scalability

Difficulty in implementation

Lost in multiple languages

Attempts from the PL Community

- Poor scalability

  + Trading off precision for scalability

  + Minimizing generated information

  - Further complicates the implementation

- Complicated implementations

  + Using declarative models such as Datalog

  - Fundamentally limited by a Datalog engine

- Lost in multiple languages

  - Nothing has been done

The Outside World

- The FB graph had **721M vertices (users), 68.7B edges (friendships)** in May 2011
- Google Maps had **20 petabytes of data** in 2015

Our "Large" Programs

FB Graph: 68.7B edges

- **The Linux kernel**, 16M lines of code; a fully inlined version has about 1B edges
- **HBase**, 1.37M lines of code; 128M edges in a fully inlined version
- **Hadoop**, 546K lines of code; 44M edges in a fully inlined version

Time for a Mindset Shift?

It is not because our programs are too large, but because we haven't thought about how to develop *scalable systems*

“Big Data” Thinking

Solution =

(1) Large Dataset + (2) Simple Computation +

System Design

Don’t complicate the algorithm

Don’t worry about too much (intermediate) data

Don’t stop at the interface between app and system

Leave the algorithm simple

Leverage modern computing resources

Design and implement *customized* systems

What We Did

Built single-machine, disk-based systems specifically for the static analysis workload

- Graspan: a graph system for CFL-reachability computation [ASPLOS'17]
- Grapple: a graph system for finite-state property checking [In Submission]

Why Systemized Static Analyses Work

- Poor scalability

No longer worry about memory blowup as we have disk-support

- Complicated implementations

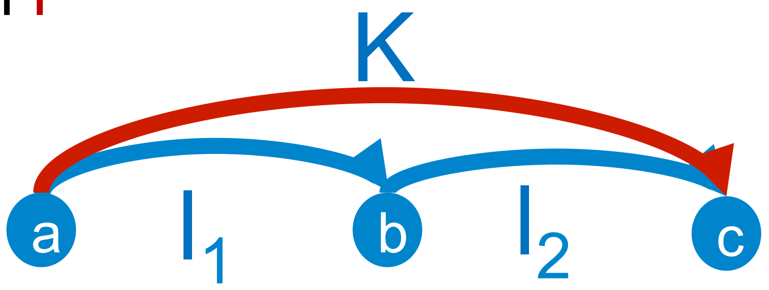
Analysis developers only implement a few interfaces; No longer worry about performance

- Lost in multiple languages

Components in different languages are turned into graphs of the same format and analyzed together

Graspan: Context-Free Language (CFL) Reachability

- A program graph P



c is K-reachable from *a*

- A context-free Grammar G with balanced parentheses properties

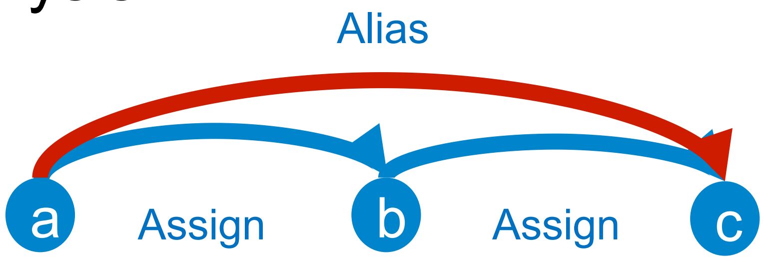
$$K \rightarrow l_1 \; l_2$$

Reps, Program analysis via graph reachability, *IST,* 1998

A Wide Range of Applications

- Pointer/alias analysis

b = a;
c = b;



Alias → Assign^+

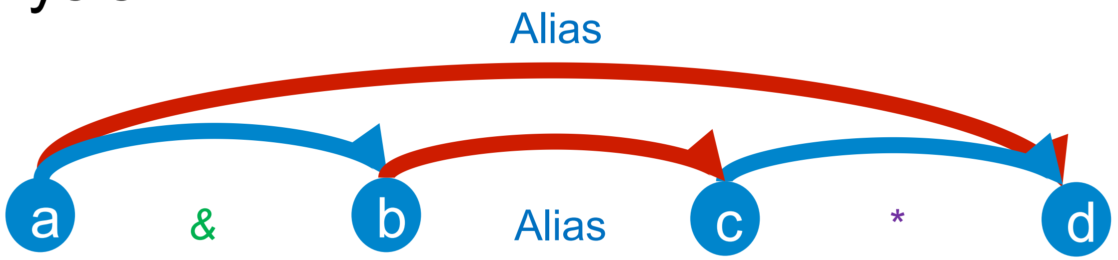
- Dataflow analysis, pushdown systems, set-constraint problems can all be converted to context-free-language reachability problems

Sridharan and Bodik, Refinement-based context-sensitive pointsto analysis for Java, *PLDI*, 2006
Zheng and Rugina, Demand-driven alias analysis for C, *POPL*, 2008

A Wide Range of Applications (Cont.)

- Pointer/alias analysis

b = & a; // *Address-of*
c = b;
d = *c; // *Dereference*



$$\text{Alias} \rightarrow \text{Assign}^{+}$$
$$\mid \;\& \;\; \text{Alias} \;\; *$$

- *Address-of &* / *dereference** are the open/close parentheses

Sridharan and Bodik, Refinement-based context-sensitive pointsto analysis for Java, *PLDI*, 2006
Zheng and Rugina, Demand-driven alias analysis for C, *POPL*, 2008

“Big Data” Thinking

Solution =

(1) Large Dataset + (2) Simple Computation +

System Design

Turning Code Analysis into Data Analytics

- Key insights:
  - The input is a fully inlined program graph
  - Adding transitive edges *explicitly* – satisfying (1)
  - Core computation is *adding edges* – satisfying (2)
  - Leveraging disk support for memory blowup
- Can existing graph systems be directly used?
  - No, none of them support dynamic addition of a lot of edges

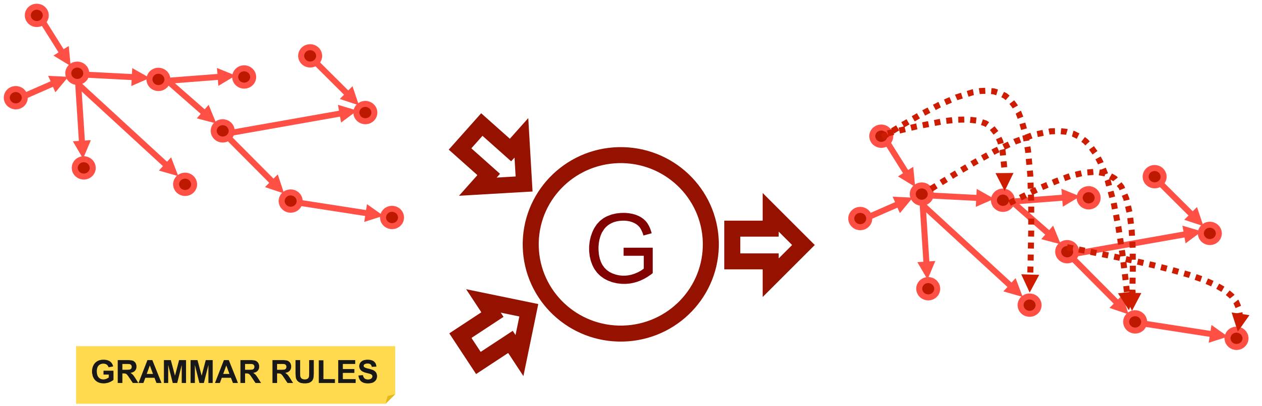
    (1) Online edge duplicate check and (2) dynamic graph repartitioning

Graspan [Wang-ASPLOS’17]

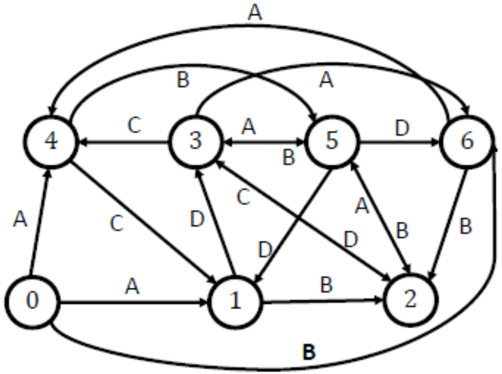
- Scalable
  - Disk-based processing on the developer's work machine
- Parallel
  - Edge-pair centric computation
- Easy to implement a static analysis
  - Implement a few interfaces

4 students + 1 postdoc, 1.5 years of development; implemented in both Java and C++
https://github.com/Graspan/

How It Works?



Granspan Design



Partition 0

Src	Dst	Label
0	1	A
	4	A
1	2	B
	3	D
2	3	C
	5	A

Partition 1

Src	Dst	Label
3	2	D
	4	C
	5	B
	6	A
4	1	C
	5	B

Partition 2

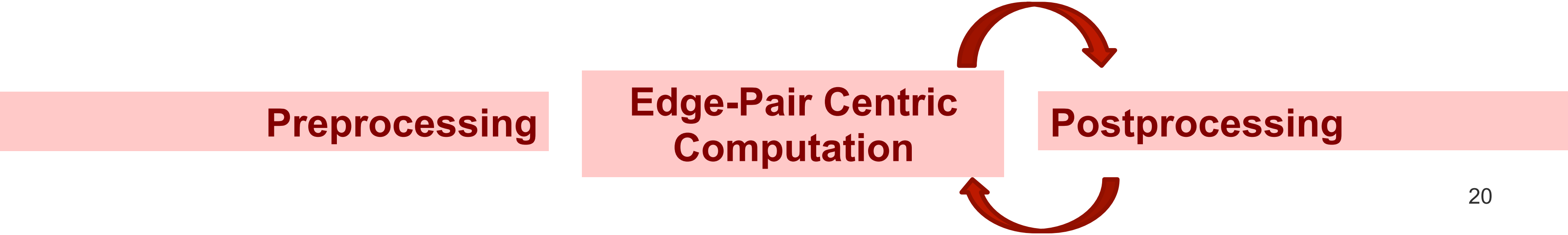
Src	Dst	Label
5	1	D
	2	B
	3	A
	6	D
6	2	B
	4	A

Preprocessing

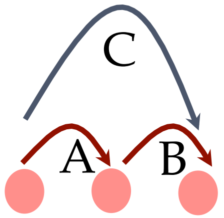
Edge-Pair Centric Computation

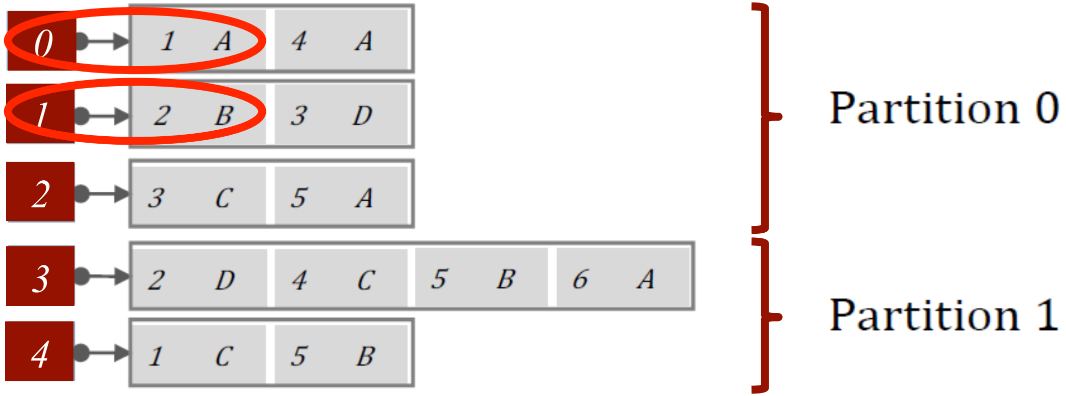
Postprocessing

Computation Occurs in Supersteps



Each Superstep Loads Two Partitions





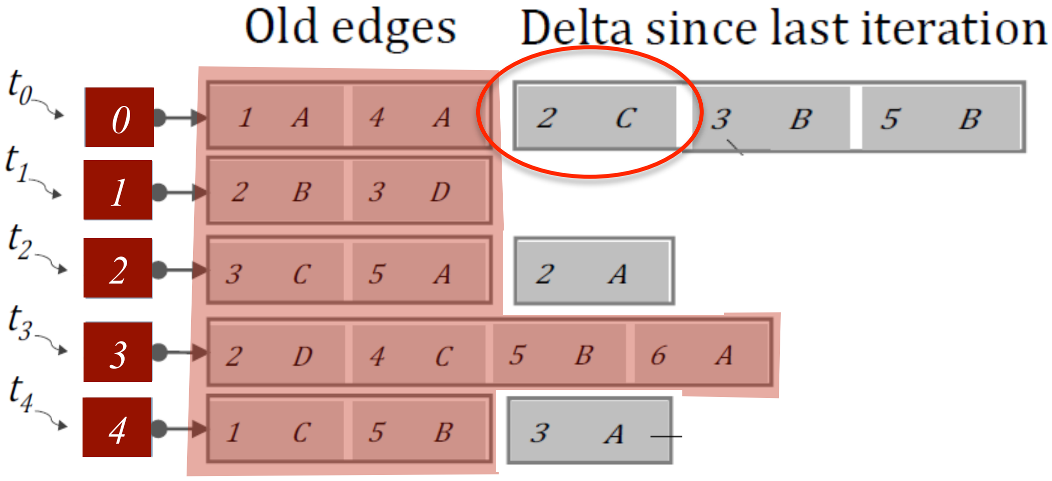
$Grammar: C := AB \quad D := BC \quad B := AD \quad A := CD$

Preprocessing

Edge-Pair Centric Computation

Postprocessing

Each Superstep Loads Two Partitions



Grammar: $C := AB$ $D := BC$ $B := AD$ $A := CD$

We keep iterating until delta is 0

Preprocessing | **Edge-Pair Centric Computation** | **Postprocessing**

Post-Processing

- Repartition oversized partitions to maintain balanced load on memory
- Save partitions to disk
- Scheduler favors in-memory partitions and those with higher *matching degrees*

Preprocessing

Edge-Pair Centric Computation

Postprocessing

What We Have Analyzed

Program	#LOC	#Inlines
Linux 4.4.0-rc5	16M	31.7M
PostgreSQL 8.3.9	700K	290K
Apache httpd 2.2.18	300K	58K

- With
  - A fully context-sensitive pointer/alias analysis
  - A fully context-sensitive dataflow analysis
- On a Dell Desktop Computer with 8GB memory and 1TB SSD

Evaluation

- Can the interprocedural analyses improve D. Englers' checkers?
  - Found 85 new NULL pointer bugs and 1127 unnecessary NULL tests in Linux 4.4.0-rc5
- How well does Graspan perform?
  - Computations took 11 mins – 12 hrs
- How does Graspan compare to other systems?
  - GraphChi crashed in 133 seconds
  - Traditional implementations of these algorithms ran out of memory in most cases
  - Datalog (SociaLite) −based implementation ran out of memory in most cases
- Will try a differential dataflow system like Naiad

Grapple: A Finite-State Property Checker

- Many bugs in large-scale systems have finite-state properties
  - Many OS bugs studied in Chou *et al.* in 2001 are finite state property bugs: *misplaced locks*, *use-after-free*, *etc.*
  - Most distributed system bugs studied in Gunawi *et al.* in 2014 are finite state property bugs: *socket leaks*, *task state problems*, *mishandled exceptions*, etc.

Gunawi *et al.*, What bugs live in the cloud? a study of 3000+ issues in cloud systems, *SoCC*, 2014
Chou *et al.*, An empirical study of operating systems errors, *SOSP*, 2001

Analyses Under the Hood

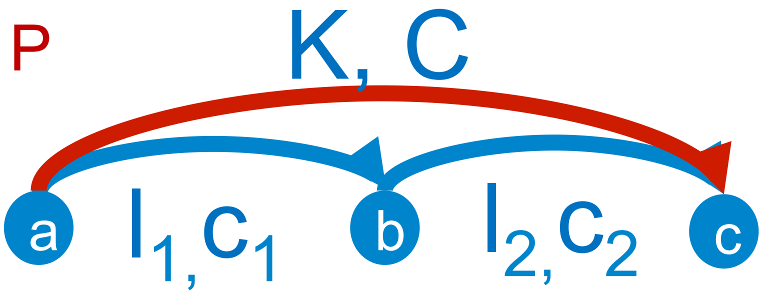
- What we need for the checker
  - Extract sequences of method calls on each object of interest
  - Check them against the FSM specification
- What analyses we need
  - Alias analysis
  - Dataflow analysis
  - Context sensitivity and path sensitivity

Grapple

- Phases
  - A fully path-sensitive, context-sensitive alias analysis
  - A fully path-sensitive, context-sensitive dataflow analysis
  - Extract event sequences
- Computation Model
  - Edge-pair-centric model
  - Challenge: how to represent and solve path constraints during graph processing

Grapple Computation Model

- A program graph P



c is K-reachable from *a*

- A context-free grammar G with balanced parentheses properties

$$K \rightarrow l_1 \; l_2$$

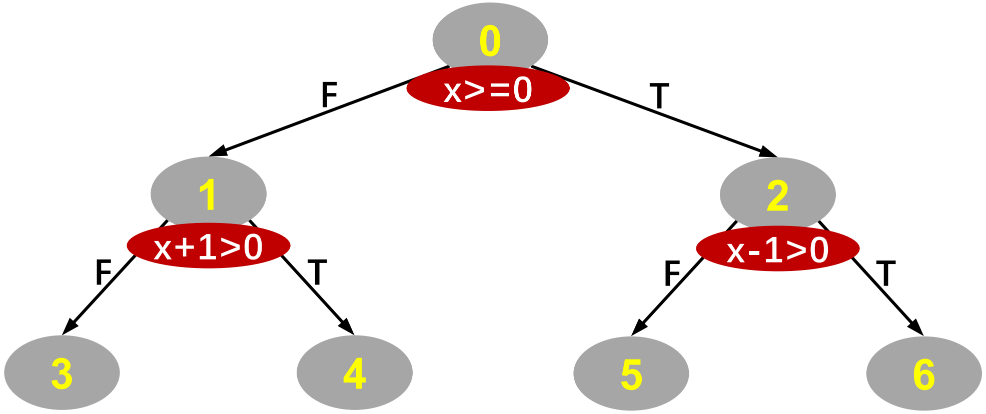
- $\mathbf{C = c_1 \wedge c_2}$ is satisfiable

Path Constraint Representation

- Challenges
  - Each edge carries only fixed-size data
  - The size is often smaller than 4 bytes
- Using *interprocedural control flow execution tree* (ICFET) as an index engine
- Each edge contains a path encoding, which is used to query for a path constraint based upon ICFET

Control Flow Execution Tree (CFET)

```
x = parse(args[0]);
y = x;                0

if(x > 0) { y--; } 2
else { y++; } 1

if(y > 0) {…} 4, 6
else {…} 3, 5
return;
```

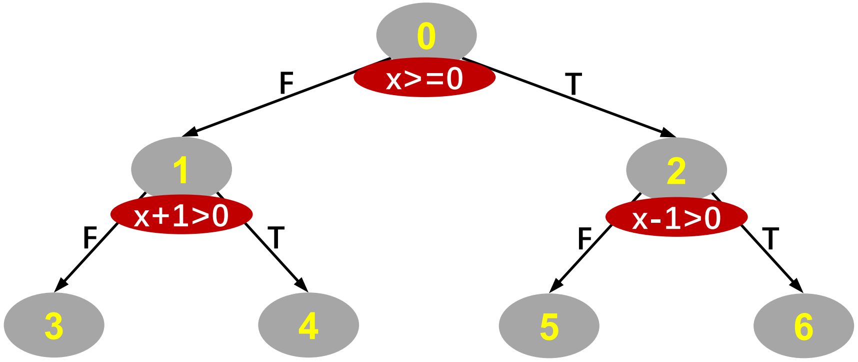


A simple numbering algorithm: **T child -> ID * 2; F child -> ID * 2 + 1**

Built before the graph computation starts

Path Representation

- An intraprocedural CFET path can be uniquely encoded as a pair **[ID$_{start}$, ID$_{end}$]**
- Decoding can be done efficiently *online*
- Loops are unrolled a certain number of times



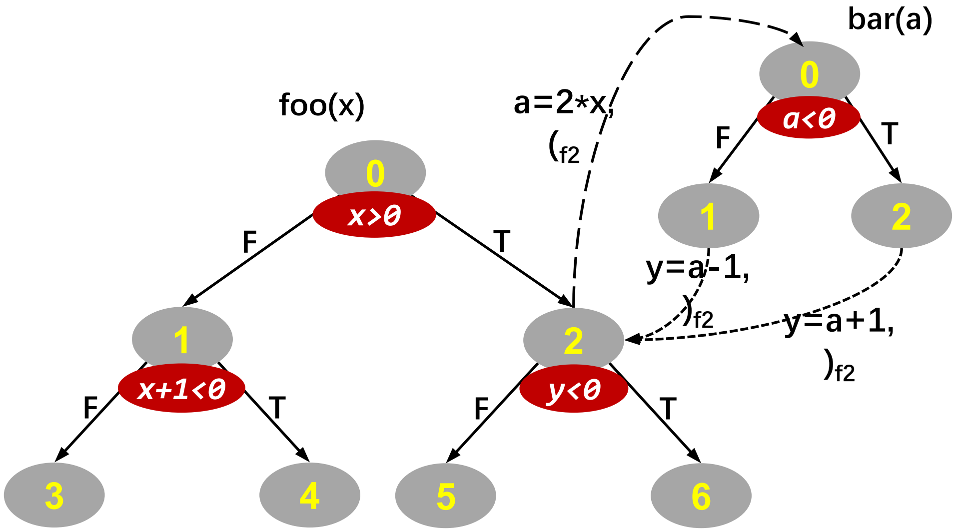
Example: [0, 6] uniquely identifies the right most path

Decoding can be done by right shifts

Symbolic execution used to compute conditions

Interprocedural CFET

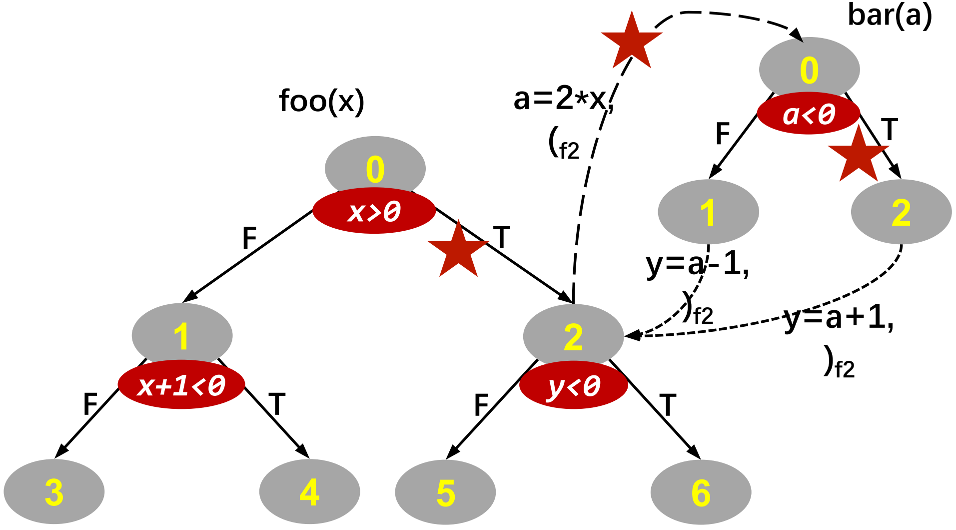
```
void foo (int x) {
  int y = x + 1;
  if (x > 0) { y = bar (2 * x); //f2 }
  if (y < 0) {…}
  return;
}

int bar (int a) {
   if (a < 0) {return a + 1;}
   return a – 1;
}
```



Connecting callers with callees using call and return edges, annotated with *call site IDs* and *symbolic equations*

Interprocedural Path Representation

- A sequence of intervals
  - [2, 0], 25, [2, 0]
  - Bounded by the call stack depth
- A constraint can be computed by extracting constraints for path fragments and combining them into a conjunctive form



Computation

- Use Graspan’s edge-pair-centric computation model
- Z3 is used for constraint solving
- Each partition is much easier to become imbalanced
  - Eager repartitioning during the computation

Evaluation Subjects

Program	#LoC	Version
Apache ZooKeeper	206K	3.5.0
Apache Hadoop	568K	2.7.5
HDFS	546K	2.0.3
Apache HBase	1.37M	1.1.6

Checkers Implemented

- IO checker
- Socket checker
- Exception handling checker
- Lock usage checker

- Checkers: 3.2K lines of Java code
- Grapple: 13K lines of C++ code, with about 1.5K lines reused from Graspan
- 1 postdoc + 5 students, 1 year of effort

Bugs Found

Checker	I/O		lock		except.		socket		total	
	RE	FP	RE	FP	RE	FP	RE	FP	RE	FP
ZooKeeper	2	0	0	0	59	0	4	0	65	0
Hadoop	0	0	0	0	54	2	0	0	54	2
HDFS	1	1	1	0	43	3	4	1	49	5
HBase	15	2	0	0	176	8	0	0	191	10

Grapple reported a total of 359 true bugs and 17 false warnings

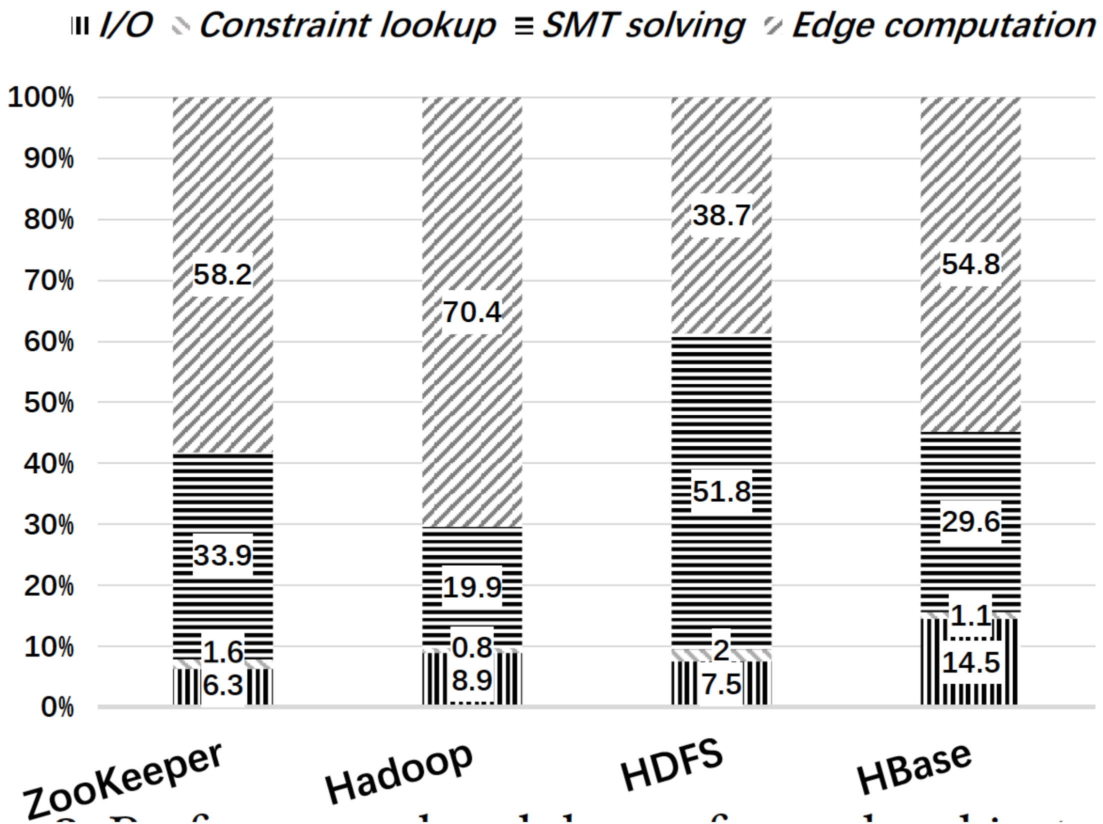
4.7% false warning rate

Grapple Performance

Subject	#V (K)	#EB (K)	#EA (K)	PT	CT	TT
ZooKeeper	6,737	12,907	20,953	1m55s	2h27m3s	2h28m58s
Hadoop	21,883	77,165	93,150	4m40s	7h24m41s	7h29m21s
HDFS	7,610	17,977	29,354	2m35s	3h37m43s	3h40m19s
HBase	44,754	128,180	202,045	19m36s	18h42m42s	19h02m18s

The execution time ranges from 2.5 hours to 19 hours

Performance Breakdown



Conclusion

- Develop systems to solve PL problems
- Try them out
  - https://github.com/graspan
  - https://github.com/grapple-system

Acknowledgements

- My (current and former) students and postdocs
  - Zhiqiang Zuo (postdoc 2015 – 2018, currently an Ass. Prof. at Nanjing University)
  - Kai Wang (Ph.D. student)
  - John Thorpe (Ph.D. student)
  - Aftab Hussain (M.S. student)

PL for Systems

Systems

I/O, Network, Computation Model, …

Existing Work

Apache Hama

Language Runtime

Memory management, compilation, hybrid memories, …

My Work

BigDatalog

Systems for PL

PL Problems

SAT Solver,
Program Analysis,
Model Checking, …

Existing Work

System Solutions

Big Data Systems

Our Work

Scalable Results

Evaluation II

- Is Graspan efficient and scalable?
  - Computations took 11 mins – 12 hrs

Prog	Pointer/Alias Analysis				
	IS=(E,V)	PS=(E,V)	PT	SS	T
Linux	(249.5M,52.9M)	(1.1B,52.9M)	91 secs	27	1.7 hrs
PSQL	(25.0M,5.2M)	(862.2M,5.2M)	10 secs	16	6.0 hrs
httpd	(8.2M, 1.7M)	(904.3M, 1.7M)	3 secs	13	8.4 hrs

Prog	Dataflow Analysis				
	IS=(E,V)	PS=(E,V)	PT	SS	T
Linux	(69.4M, 63.0M)	(211.3M, 63.0M)	65 secs	33	11.9 hrs
PSQL	(34.8M,29.0M)	(56.1M, 29.0M)	35 secs	16	2.4 hrs
httpd	(10.0M, 5.3M)	(19.3M, 5.3M)	9 secs	16	11.4 mins

Evaluation III

- Graspan v/s other engines?
  - GraphChi crashed in 133 secs

Analysis	Graspan		ODA [101]	SociaLite [45]
	CT	I/O		
Linux-P	99.7 mins	46.6 secs	OOM	OOM
Linux-D	713.8 mins	4.2 mins	-	OOM
PostgreSQL-P	353.1 mins	4.5 mins	> 1 day	OOM
PostgreSQL-D	143.8 mins	57.1 secs	-	OOM
httpd-P	497.9 mins	7.6 mins	> 1 day	> 1 day
httpd-D	11.3 mins	3.3 secs	-	4 hrs

[101] X. Zheng and R. Rugina, Demand-driven alias analysis for C, POPL, 2008
[45] M. S. Lam, S. Guo, and J. Seo. SociaLite: Datalog extensions for efficient social network analysis. ICDE, 2013.

Program Graph Generation

```
x = parse(args[0]);  y = x;

FilterWriter out = null, o = null;

if(x > 0) {
   out = new FilterWriter();
   o = out;
   y--;
}
else { y++; }

if(y > 0) {out.write(…); o.close();}

return;
```