Graspan

A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code

ASPLOS 2017

Kai Wang - Aftab Hussain - Zhiqiang Zuo - Guoqing Xu - Ardalan Amiri Sani
University of California, Irvine

Bugs are everywhere

Bugs are everywhere

Ariane 5
Disaster
1996

NE America
Blackout
2003

Bugs are everywhere

Ariane 5
Disaster
1996

NE America
Blackout
2003

NASA Mariner 1
1962

USS Yorktown
Incident
2007

Bugs are everywhere

Toyota recalls
Since 2009

Ariane 5
Disaster
1996

NE America
Blackout
2003

NASA Mariner 1
1962

USS Yorktown
Incident
2007

Bugs are everywhere

$312B Globally
$60B US

Toyota recalls
Since 2009

Ariane 5
Disaster
1996

Scalability

We scale **context-sensitive inter-procedural analysis** for **bug detection tools**

We turn the power of **big data analysis** in **bug detection tools**

Huge Software

Analysis Technique

We build a **big data infrastructure**


Problem
Graspan
Results

Problem

Why Interprocedural Analysis?

NULL Bug Checker

* Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller, **Faults in linux: ten years later,** ASPLOS '11

* Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler, **An empirical study of operating systems errors,** SOSP `01

```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = fn_explct_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
}
```

Reported **98** bugs in **Linux 2.6.1**

Reported **20** bugs in **Linux 4.4.0**, all of which were false positives

```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = fn_explct_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
}
```

Absence of this check is a bug.

```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = function2();
  }
  else {
    ptr1 = fnA();
  }

    int b = *ptr1;

}
```

```
int * function2() {
   int * ptr2 = NULL;
   int * ptr3;
   if (<condition>) {
      ptr3 = ptr2;
   }
   else {
      ptr3 = fnB();
   }
   return ptr3;
}
```

Null checker has no way of knowing that ptr1 can be NULL.

Augmenting Null Checker with this info. can help us find **more bugs**.

Reported **85 new bugs** in Linux 4.4.0.

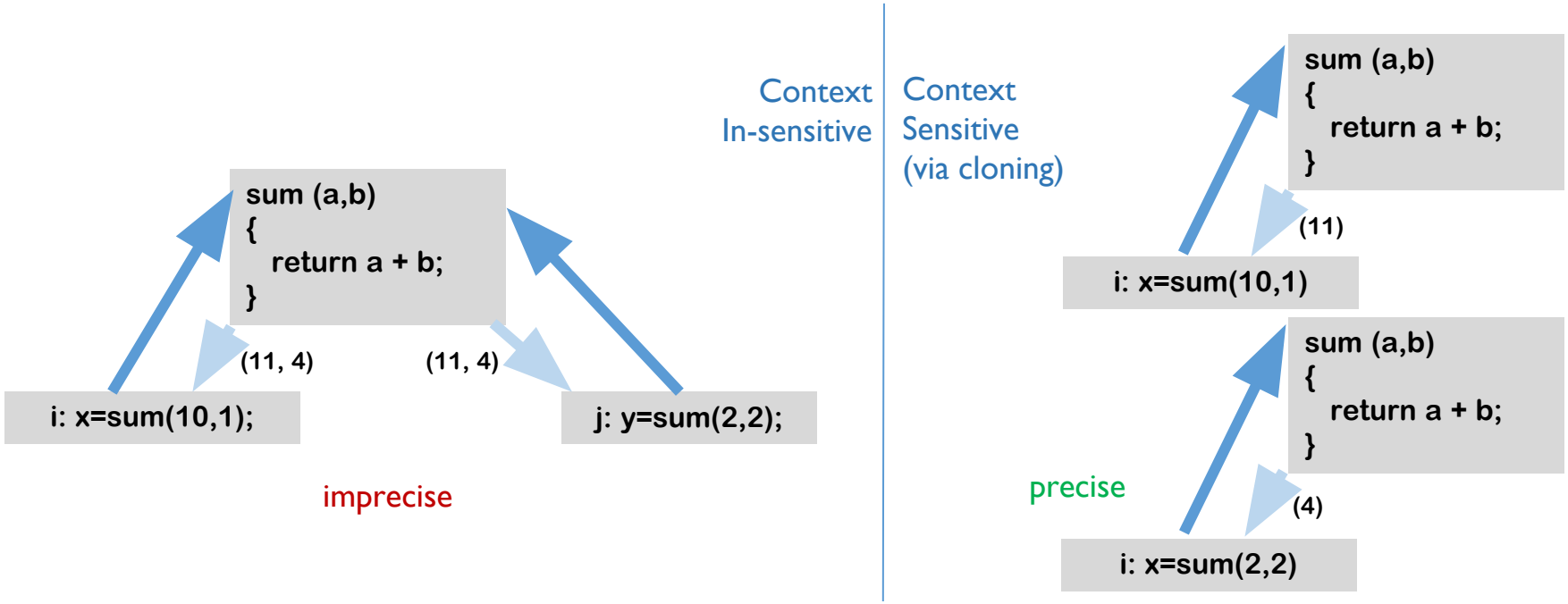
```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = function2();
  }
  else {
    ptr1 = fnA();
  }

    int b = *ptr1;

}
```

```
int * function2() {
   int * ptr2 = NULL;
   int * ptr3;
   if (<condition>) {
      ptr3 = ptr2;
   }
   else {
      ptr3 = fnB();
   }
   return ptr3;
}
```

A dataflow analysis tells us that ptr1 can be null.

Scalability Problem
in
Context-Sensitive
Interprocedural Analysis (CSIA)

The scalability problem of CSIA stems from producing distinct solutions for different <u>calling contexts</u>.



No. of calling contexts grows **exponentially** with program size.

A moderate-sized program can have **10^{14} distinct contexts**.

* John Whaley and Monica S. Lam,
Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, PLDI '04

What do we do?

Parallelization?

Techniques are information discovery based.

Approximation?

Implementing the approximations is complicated.

In Sridharan`s and Bodik`s work, **more than 75%** of their entire code was dedicated to tuning the analysis.

* Manu Sridharan and Rastislav Bodík, **Refinement-based context-sensitive points-to analysis for Java,** PLDI ‘06

Our Goal

Peform **precise analysis (like fully CSIA)** efficiently

Make program analysis more **parallelizable** and **scalable**

We want the analysis implementation to be **simple**

Graspan

Scalable — Disk-based processing in developer`s work machine

Developer enjoys **high precision** without worrying about **scalability** or **efficiency.**

Parallelizable — Edge-Pair Centric Computation Model

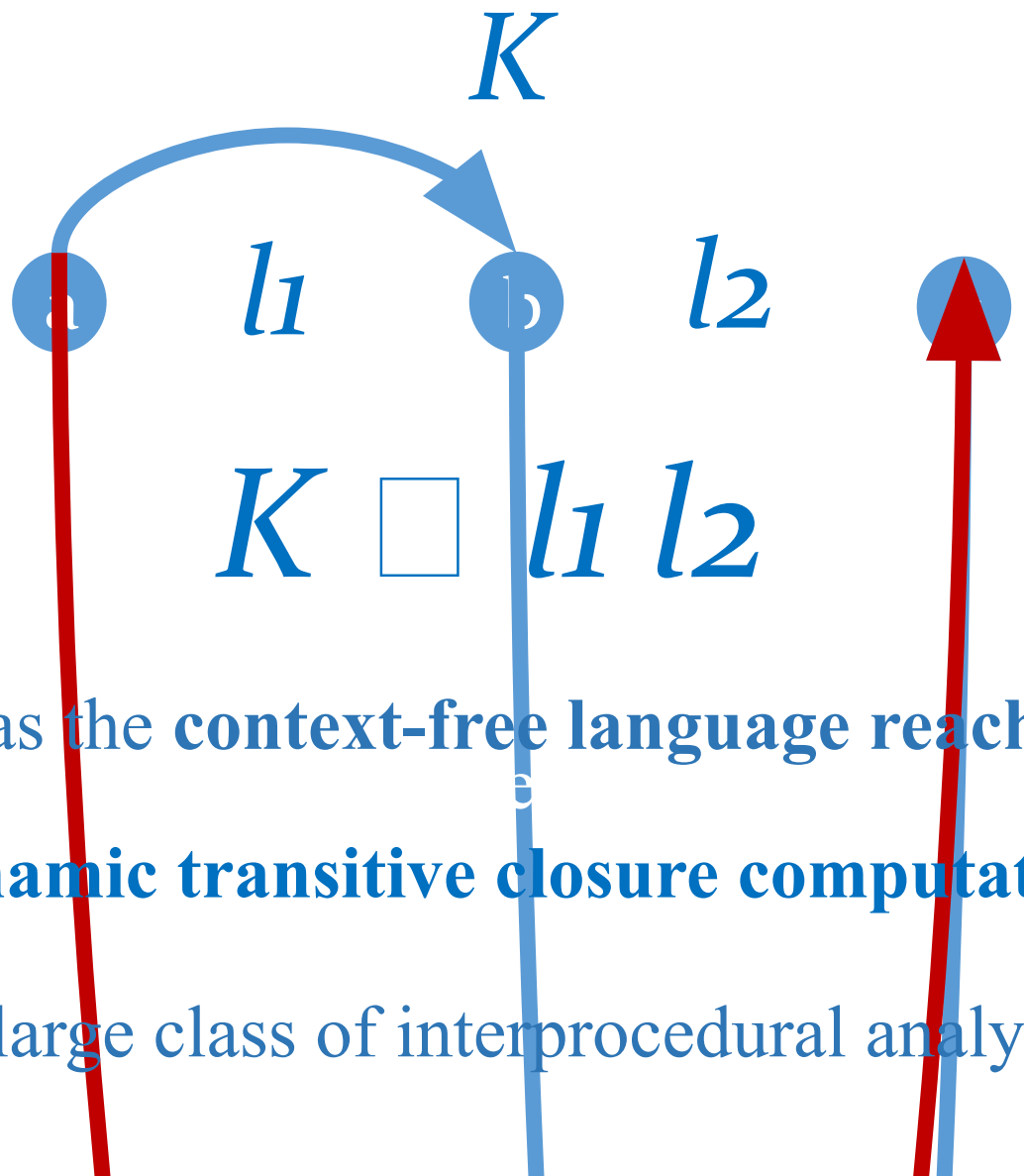
Makes implementing program analysis easy — Developer needs to provide a context-free grammar

Graspan

Program Analysis and Graphs?

Thomas Reps et al. showed most interprocedural analyses, like **dataflow analysis** and **pointer analysis**, can be transformed to a **graph reachability problem**

* Thomas Reps, Susan Horwitz, and Mooly Sagiv, **Precise interprocedural dataflow analysis via graph reachability**, POPL '95



This is known as the **context-free language reachability problem**

dynamic transitive closure computation

Can solve a large class of interprocedural analysis problems

How did we use Graspan?

Using Graspan, we have implemented fully
context-sensitive
pointer analysis and **dataflow analysis**.

Input graphs

For **pointer analysis**, we generated a program expression graph.

Zheng and Rugina, **Demand-driven alias analysis for C**, POPL `08

For **dataflow analysis**, we generated an exploded supergraph.

Thomas Reps, Susan Horwitz, and Mooly Sagiv, **Precise interprocedural dataflow analysis via graph reachability**, POPL '95

Input graphs

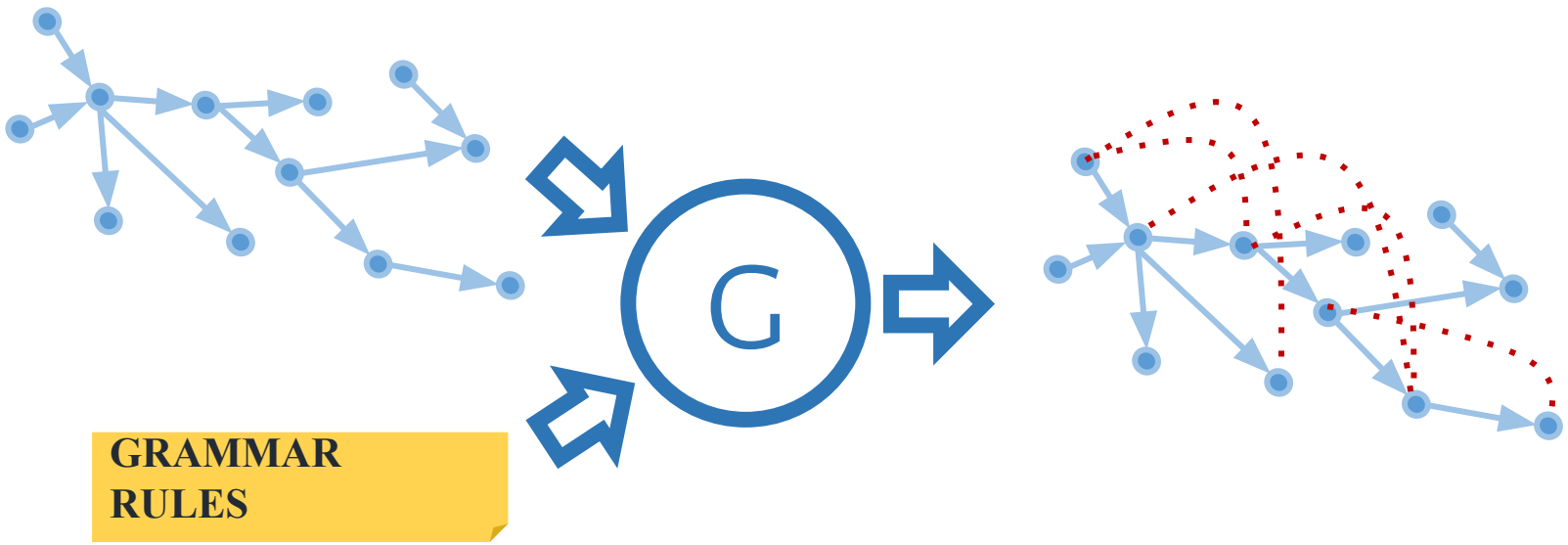
To achieve context sensitivity, we performed **bottom up inlining** on the call graphs.

We clone a function`s entire graph for each call site that calls the function.

Input graphs

Recursive functions are handled
context-insensitively.

How it works



Our Design

Preprocessing

Edge-Pair Centric Computation

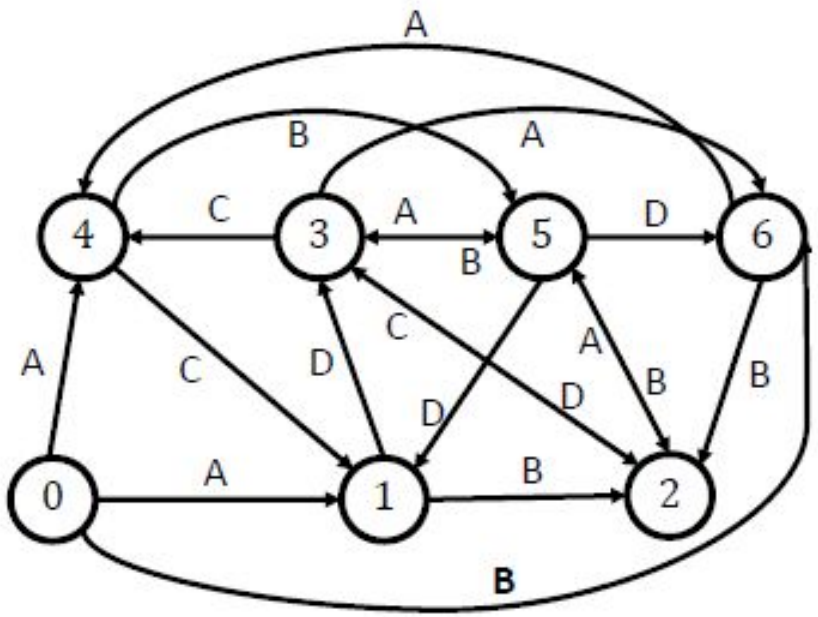
Post-Processing

Generates partition files from the graph and stores them on disk.

Preprocessing

Edge-Pair Centric Computation

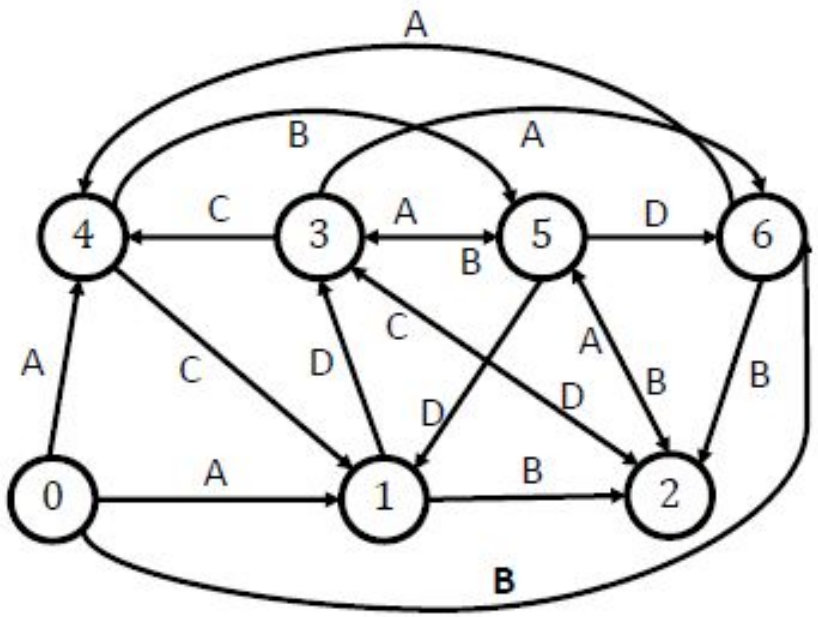
Post-Processing



Partition 0			Partition 1			Partition 2		
Src	Dst	Label	Src	Dst	Label	Src	Dst	Label
0	1	A	3	2	D	5	1	D
	4	A		4	C		2	B
1	2	B		5	B		3	A
	3	D		6	A		6	D
2	3	C	4	1	C	6	2	B
	5	A		5	B		4	A

Edges with same source id are in the same partition.
Helps to keep track of unique edges.

Preprocessing | **Edge-Pair Centric Computation** | **Post-Processing**



Partition 0			Partition 1			Partition 2		
Src	Dst	Label	Src	Dst	Label	Src	Dst	Label
0	1	A	3	2	D	5	1	D
	4	A		4	C		2	B
1	2	B		5	B		3	A
	3	D		6	A		6	D
2	3	C	4	1	C	6	2	B
	5	A		5	B		4	A

Partitions are of similar size.

Keep a balanced load on the memory.

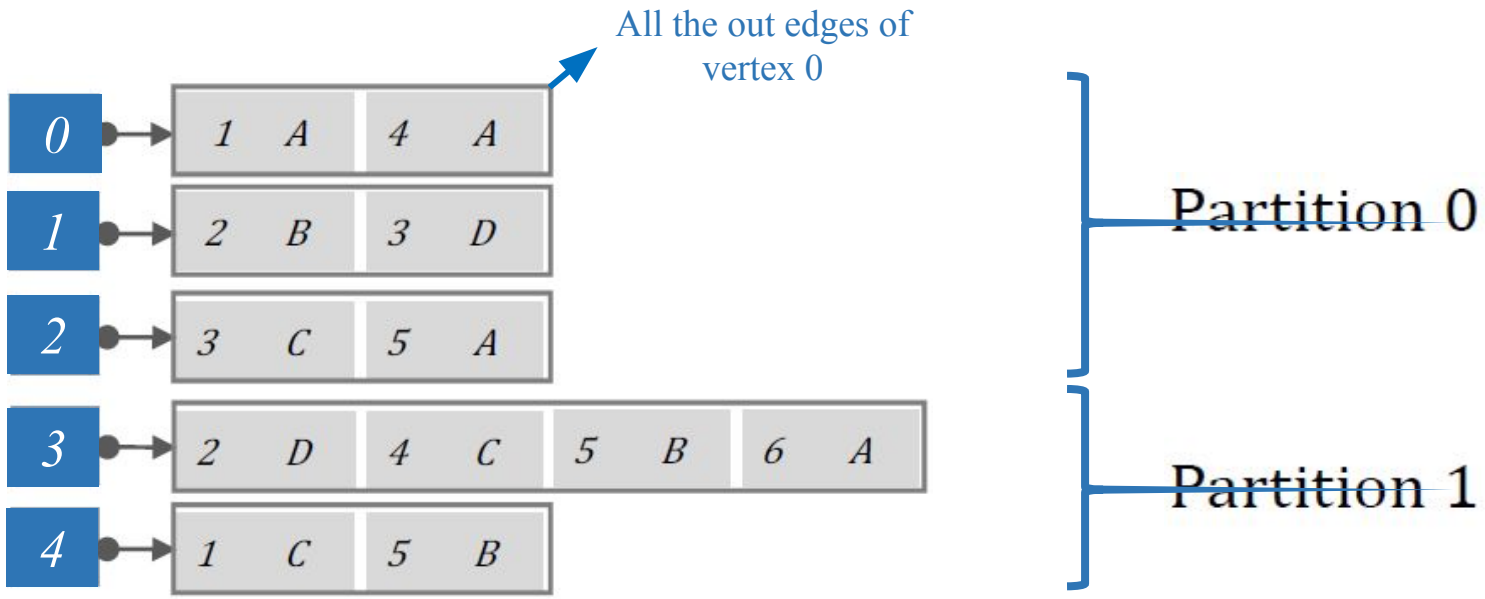
Preprocessing | **Edge-Pair Centric Computation** | **Post-Processing**

In each superstep

A **scheduler** selects two partitions to load from disk to memory for computation.

After computation, the partitions are saved to disk,
or one is retained for the next superstep.

The process repeats until there are no more new edges added globally.

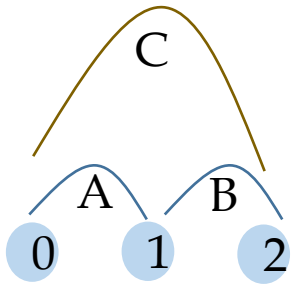


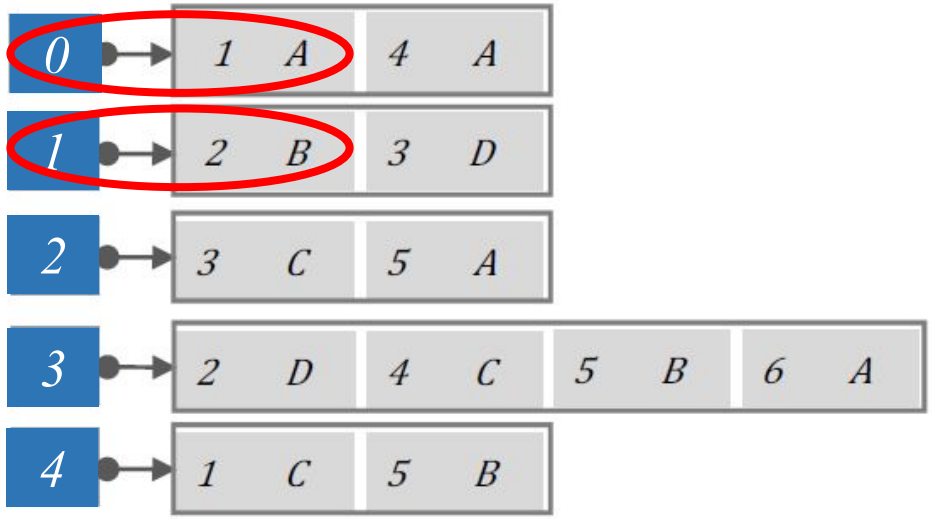
$$Grammar: \quad C := AB \quad D := BC \quad B := AD \quad A := CD$$

Preprocessing

Edge-Pair Centric Computation

Post-Processing





Edge-matching

Find consecutive pairs of edges, whose label satisfies grammar.

Add a new edge if such a match is found, unless it already exists.

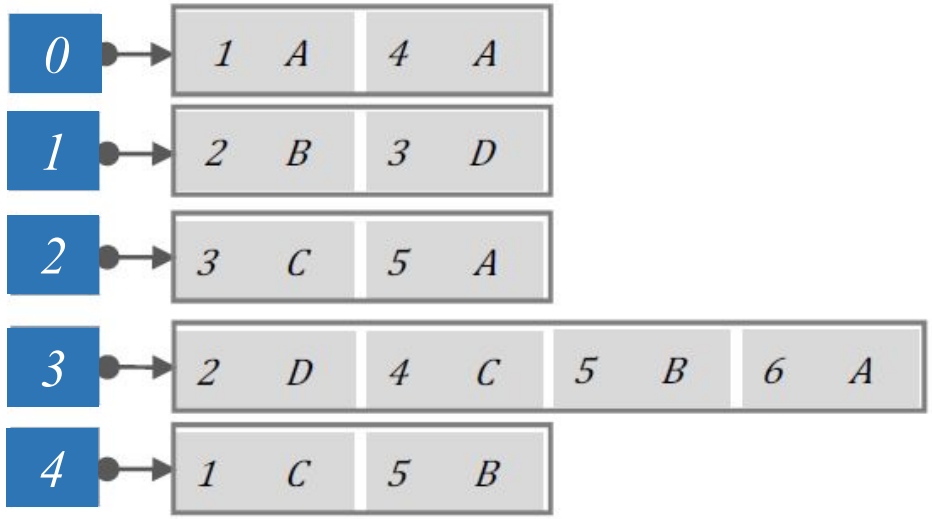
$$Grammar: C := AB \quad D := BC \quad B := AD \quad A := CD$$

Preprocessing

Edge-Pair Centric Computation

Post-Processing

Edge-matching



Straight-forward way:
For each edge (***a***,***b***), check all of ***b***`s outgoing edges.

$O(|E|^2)$

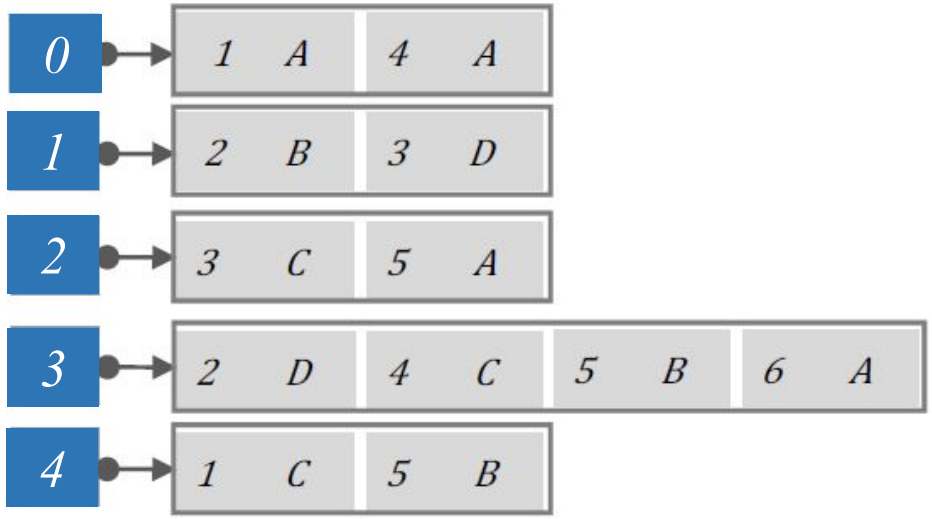
$$Grammar: \quad C := AB \quad D := BC \quad B := AD \quad A := CD$$

Preprocessing

Edge-Pair Centric Computation

Post-Processing

Edge-matching



Our Approach:
We join the lists using a MinHeap based algorithm, until there are no more edges added.

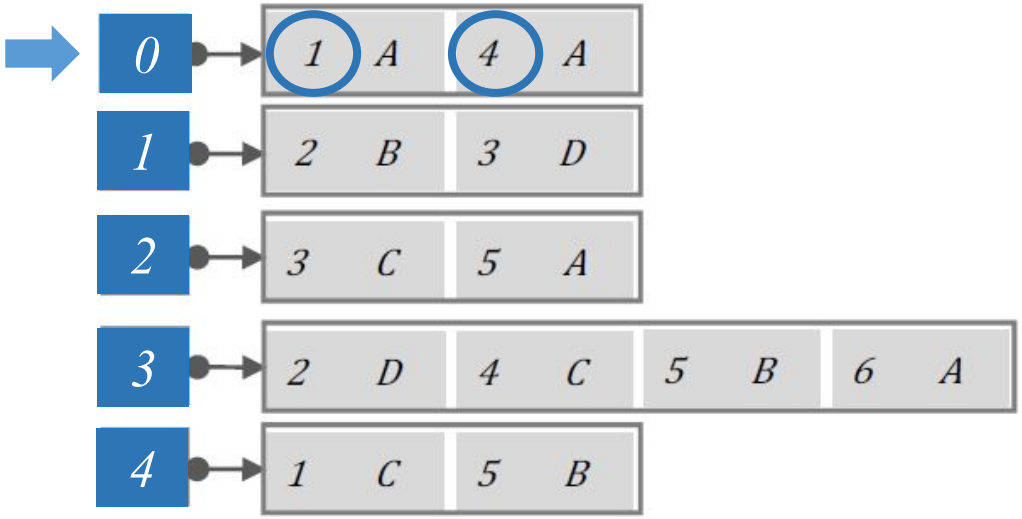
$O(|E|\log|V|)$

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

Preprocessing

Edge-Pair Centric Computation

Post-Processing

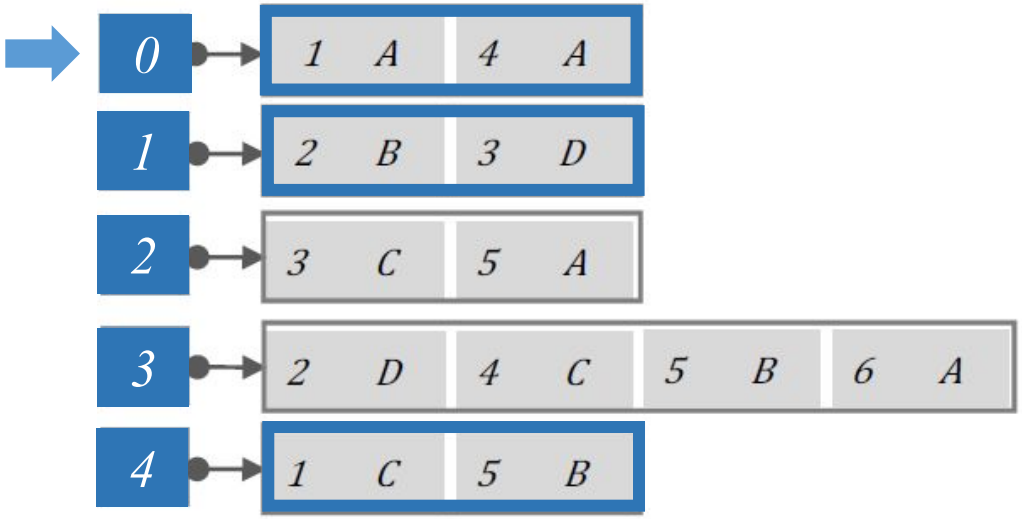


$$Grammar: \quad C := AB \quad D := BC \quad B := AD \quad A := CD$$

Preprocessing

Edge-Pair Centric Computation

Post-Processing

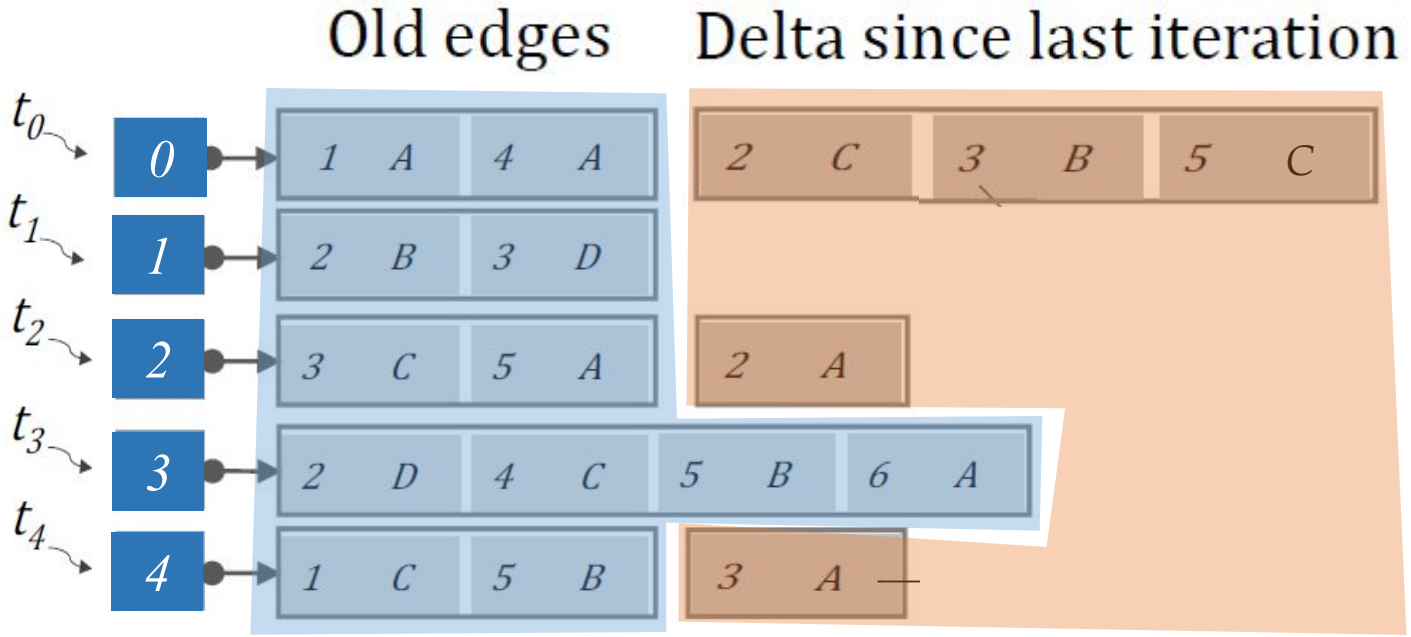


$$Grammar: \quad C := AB \quad D := BC \quad B := AD \quad A := CD$$

Preprocessing

Edge-Pair Centric Computation

Post-Processing



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

We keep iterating until delta is 0.

Preprocessing

Edge-Pair Centric Computation

Post-Processing

Repartition oversized partitions
to maintain balanced load on memory.

Saves partitions to disk.

Preprocessing

Edge-Pair Centric Computation

Post-Processing

Results

What we 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

GRAPH SIZES (#Edges)	Points-to graph	Dataflow graph
Linux	229.3M	49.5M
PostgreSQL	177.3M	219.8M
Apache httpd	4.5M	4.8M

Results

The machine we used

Dell Desktop Computer
Quad-Core 3.2GHZ Intel i5-4570 CPU
8GB Memory
1TB SSD
Linux 4.2.0

Research Questions

Can interprocedural Analysis improve Engler`s Checkers?

Found 85 new Null Dereference bugs in Linux

How easy was it to use Graspan?

1K LOC of C++ for writing each of points-to and dataflow graph generators. Provide a grammar file.

Is Graspan Efficient and Scalable?

Computations took 3 – 5 hrs

Graspan v/s existing backend engines?

GraphChi (a graph system) crashed in 133 secs.
SocialLite (a Datalog engine) ran out of memory.

Future Ambitions

Extend system support for
other program analysis tasks like (path-sensitive
analysis and constraint-based analyses)

Extend the vision of this system to support
efficient execution of Datalog programs.

THANK YOU

Graspan

Evaluation

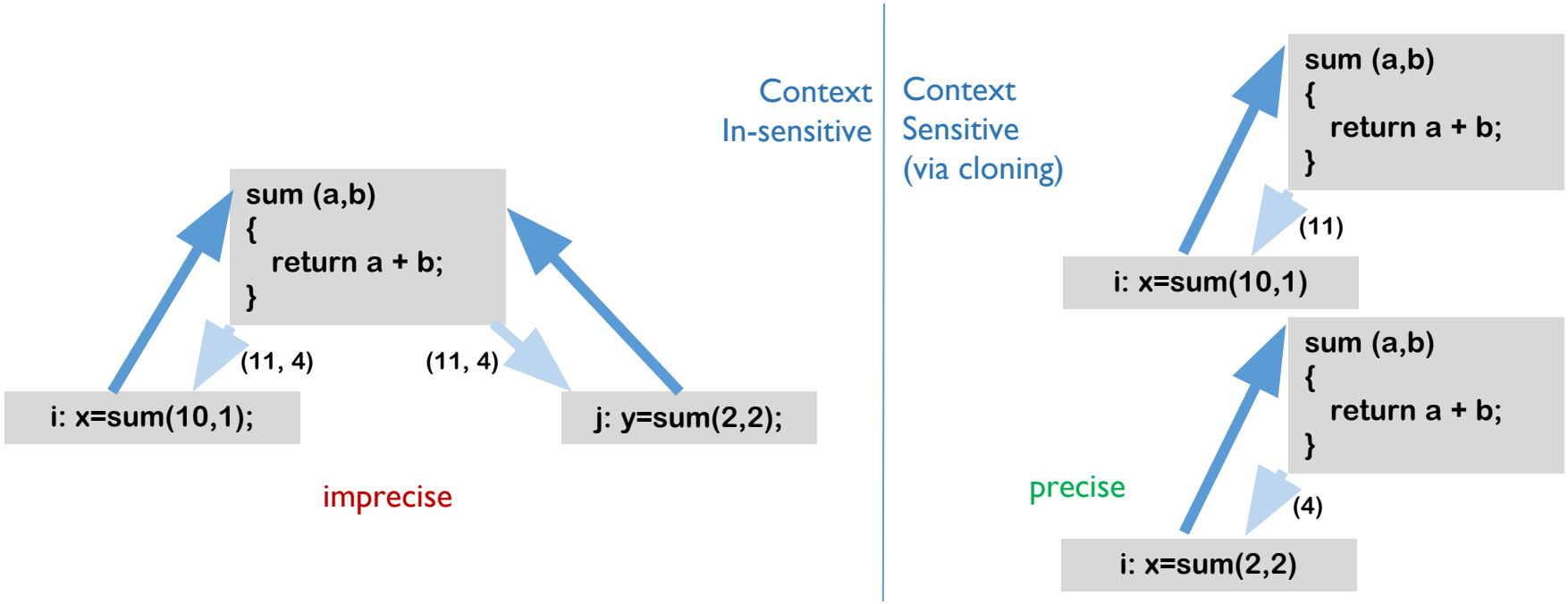
Results

Example Bug

```
void*probe_kthread_data(
    task_struct *task){
 void *data = NULL;
 probe_kernel_read(&data);

 /*data will dereferenced
 after return.*/
 return data;
}

long probe_kernel_read
 (void *dst){
 if(...)
   return -EFAULT;
 return
 __probe_kernel_read(dst);
}
```

(NULL deref in kernel/kthread.c)

The scalability problem of interprocedural analysis stems from producing distinct solutions for different **calling contexts**.



NULL BUG CHECKER [1]

```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = fn_explct_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
}
```

NULL BUG CHECKER [1]

```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = fn_explct_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
}
```

NULL BUG CHECKER [1]

```
void function1() {
  int * ptr1;
  if (<condition>) {
    ptr1 = fn_explct_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
}
```

Intraprocedural Static Analysis

Lessons Learned

- Most of the bugs they can catch have already been fixed.

coverage

execution

Dynamic	Static

Static

Intraprocedural

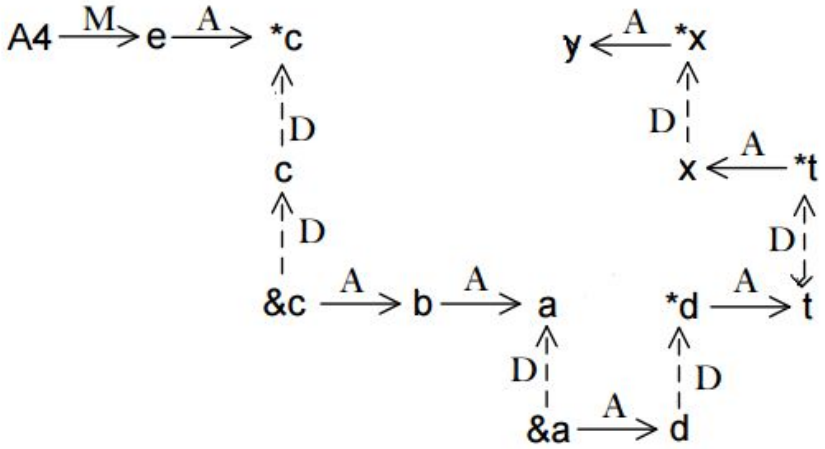
Interprocedural

Intraprocedural Static Analysis

Graspan

How to use it

```
1 a = b;
2 b = &c;
3 d = &a;
4 e = malloc(...);
5 *c = e;
6 t = *d;
7 x = *t;
8 y = *x;
```



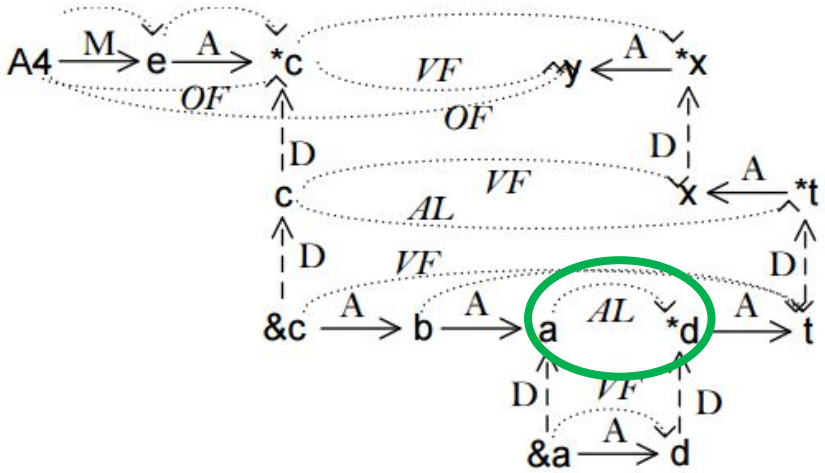
+

Grammar Rules

Object flow:	*objectFlow*	::=	M *valueFlow*
Value flow:	*valueFlow*	::=	$(A$ *alias*?$)^*$
Expr alias:	*alias*	::=	$\overline{D}$ *valueFlow* D

Input

Output



Intraprocedural Static Analysis

Individual functions are analyzed in isolation.

Most intraprocedural static bug detection techniques use pattern matching.

Static Analysis

Analyzing code without executing it

Useful for bug finding

Most existing techniques use pattern matching

Pattern matching static analysis techniques

Simple, easy to implement

Heuristic based and can miss deep bugs

Static Analysis

Analyzing code wi

Useful for bug find

Most existing tech

Pattern matching static analysis techni

Simple, easy to implem

Heuristic based and car

If any pointer variable = functionA, check ptr

```
int * fn1 () {
  int * ptr1;
  if (<a condition>) {
    ptr1 = fn_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
return ptr1;
}

int * fn2() {
   int * ptr2 = fn1();
   int c = *ptr2;
   return ptr2;
}
```

Such techniques have been found to miss bugs .

Static Analysis

Sophisticated static analysis techniques are not scalable.

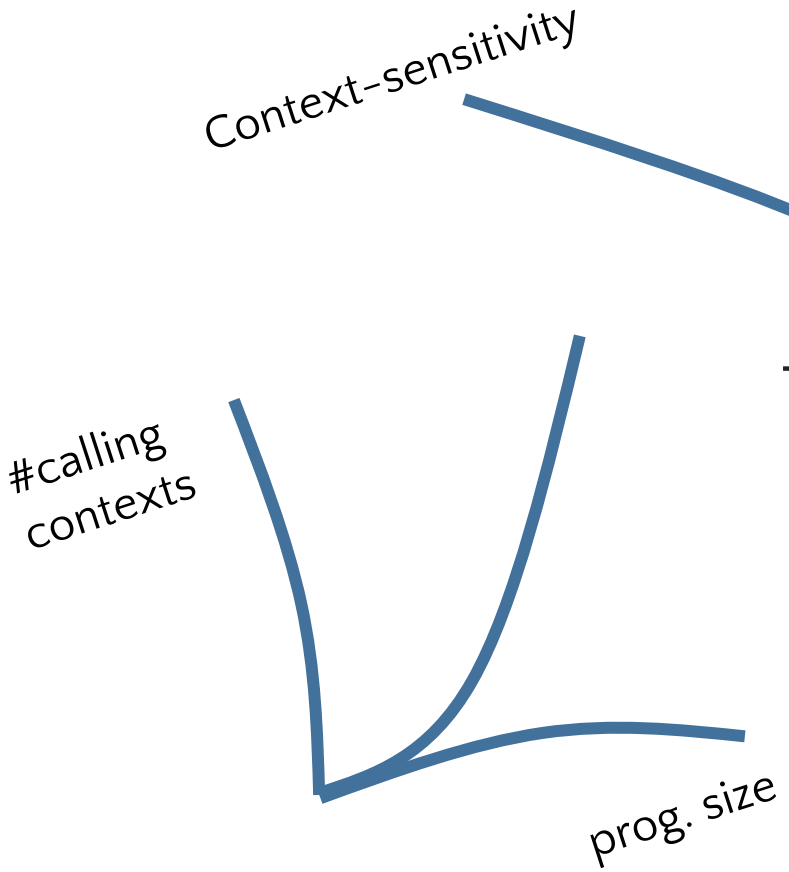
They are **computation intensive** and **difficult to program**.

Static Analysis

Context-sensitivity

They are **computation intensive** and **difficult to program**.

Static Analysis



They are **computation intensive** and **difficult to program**.

Static Analysis

Implementing heuristics to perform approximations

They are **computation intensive** and **<u>difficult to program</u>**.

Intraprocedural Static Analysis

Pattern matching

* Nicolas Palix, Julia Lawall, and Gilles Muller. 2010. **Tracking code patterns over multiple software versions with Herodotos.** In Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10).

* Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. **Bugs as deviant behavior: a general approach to inferring errors in systems code**. In Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)

* Static source code analysis, static analysis, software quality tools by **Coverity** Inc. http://www.coverity.com/, 2008.

* Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. **Faults in linux: ten years later.** SIGARCH Comput. Archit. News 39, 1 (March 2011), 305-318.

* D. R. Engler, B. Chelf, A. Chou, and S. Hallem. **Checking system rules using system-specific, programmer-written compiler extensions.** In Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 1–16, San Diego, CA, Oct. 2000.

* J. L. Lawall, J. Brunel, R. R. Hansen, H. Stuart, G. Muller, and N. Palix. WYSIWIB: **A declarative approach to finding protocols and bugs in Linux code.** In The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, (DSN 2009), pages 43–52, Estoril, Portugal, June 2009.

* D. Wheeler. **Flawfinder** home page. Web page: http://www.dwheeler.com/flawfinder 2006.

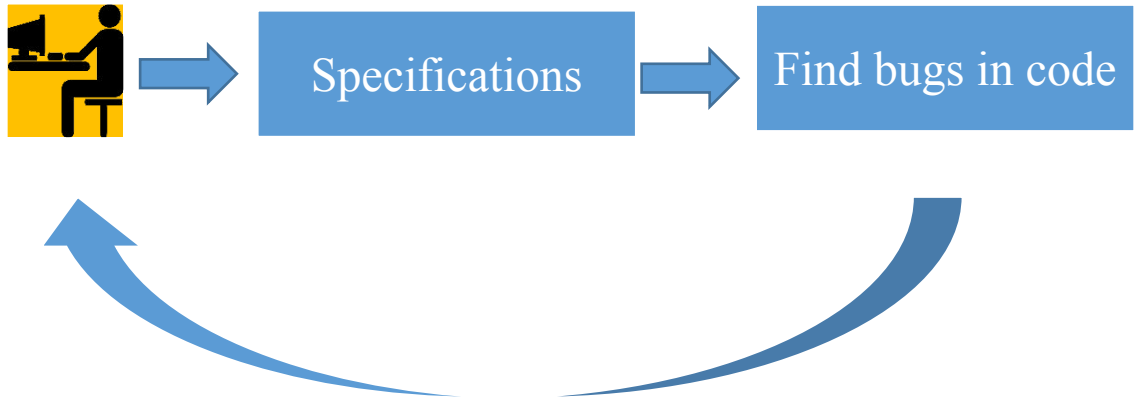
* David Hovemeyer and William Pugh. 2004. **Finding bugs is easy**. *SIGPLAN Not.* 39, 12 (December 2004), 92-106.

Pattern Matching Works

J. L. Lawall et al.
WYSIWIB: A declarative approach to finding protocols and bugs in Linux code
DSN 2009

WYSIWIB: “what you see is where it bugs”

Coccinelle



Palix, et al. Faults in linux: ten years later
ASPLOS 2011

Used the tool to find these bugs in Linux versions 2.6.0 to 2.6.33,

Deadlocks, Null Pointer Dereferences, Double use of freed objects, etc.

Fault Rate and Fault Distribution across Linux directories.

Pattern Matching Works

Engler, et al.
Checking system rules using system-specific, programmer-written compiler extensions.
OSDI 2000

Meta-level compilation

Use compiler extensions in a language called metal

Extend *xg*++ compiler (based on *g*++)

Found 500 errors (block, double lock, double unlock, etc.) in
Linux 2.3.99, OpenBSD, Xok exokernel, FLASH machine`s embedded
cache controller

Their extensions are less than 100 lines of code.

Pattern Matching Works

Engler et al.
Bugs as deviant behavior: a general approach to inferring errors in systems code.
SOSP '01

What are the rules?

E.g.
Spin_lock(a) followed by **Spin_unlock(a)**
999 times out of 1000

The single instance without this pattern is a bug.

User asks queries using templates.
Does lock <L> protect variable <V>?

They generate,
E - #uses of V protected by L
V - #uses of V

The queries are ranked using z(E,N)

Using the rules generated they implemented 6 kinds of bug checkers, in *metal*.

Found errors in Linux 2.4.1 and 2.4.7

Intraprocedural Static Analysis

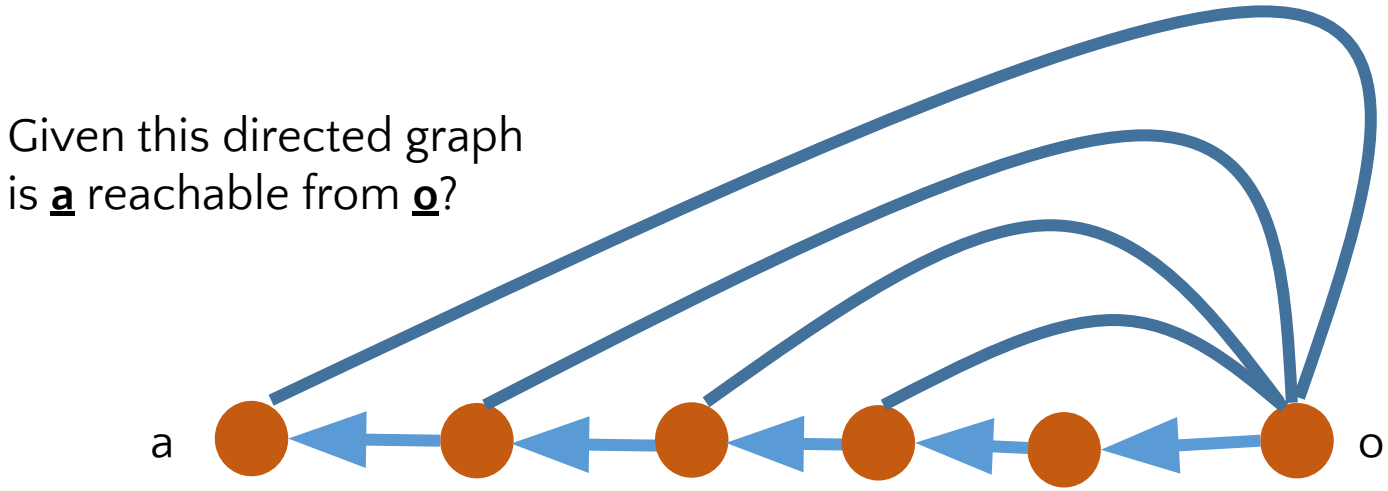
Lessons Learned

+ These techniques are easy to implement.

- They can`t find deeper bugs.

Our Observation

Many formulations in Interprocedural analysis can be formulated as a graph reachability problem. *(Thomas, Reps Program Analysis via Graph Reachability, 1998)*

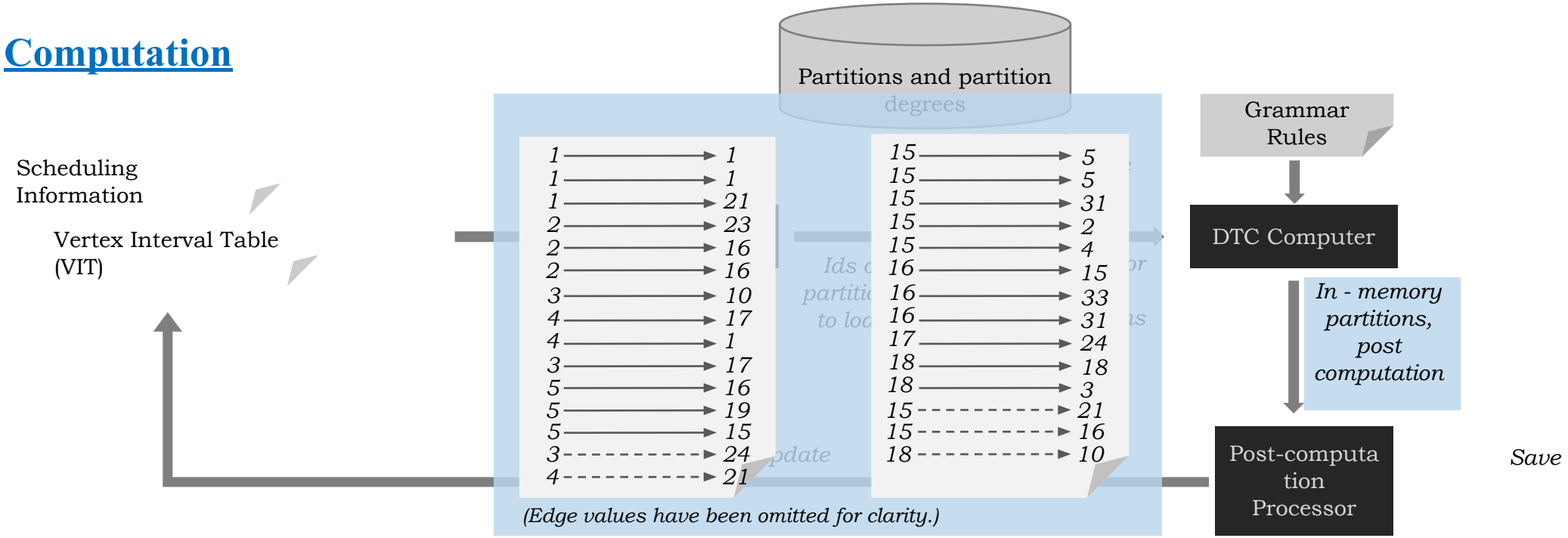


This process of dynamically adding edges is known as **dynamic transitive closure computation**.

Graspan

Overall System Design

Computation



Graspan

Loading

In memory Data Structures

(a) Adjacency lists for each loaded partition.

Partition: *1 (edges with source vertices from 1 to 4)*

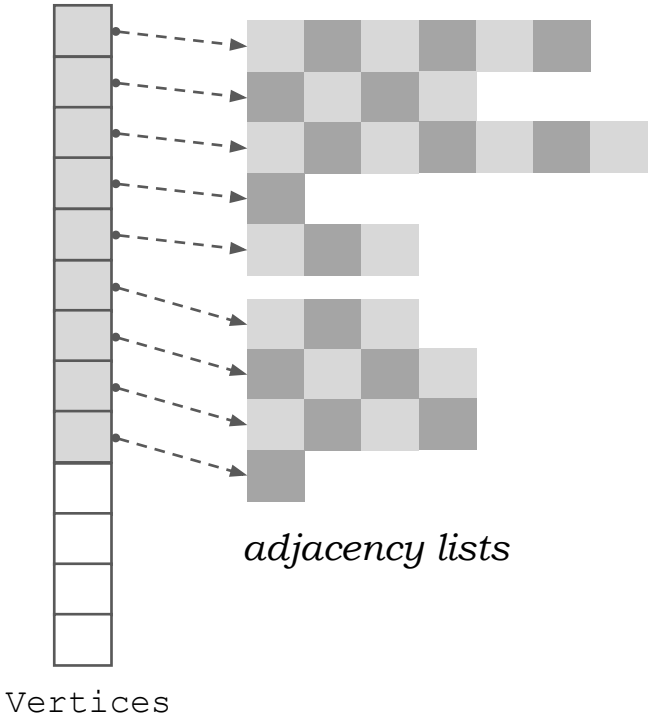
(1)	*12*	*13*	*14*	*15*	*15*	*8*
(2)	*3*	*9*	*11*			
(3)	*9*	*1*	*46*	*46*	*50*	
(4)	*11*	*18*	*31*	*10*		

source vertex id

destination vertex id of existing edge

Similar corresponding adjacency lists are maintained for edge values.

(b) Vertices data structure to refer to the adjacency list of each loaded source vertex.



Graspan

Parallel EP-centric Computation Algorithm

	old	new	old U new	delta	old U new U delta
#1	<EMPTY>	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24

Graspan

Parallel EP-centric Computation Algorithm

	old	new	old U new	delta	old U new U delta
#1	<EMPTY>	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24
#2	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24		

Graspan

Parallel EP-centric Computation Algorithm

	old	new	old U new	delta	old U new U delta
#1	<EMPTY>	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24
#2	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24		

Graspan

Parallel EP-centric Computation Algorithm

	old	new	old U new	delta	old U new U delta
#1	<EMPTY>	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24
#2	1: **2**,**3**,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: **2**,**3**,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24		

Graspan

Parallel EP-centric Computation Algorithm

	old	new	old U new	delta	old U new U delta
#1	<EMPTY>	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24
#2	1: **2,3,**10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,**4,5**,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 **4**: 5,17,18,20,21,22,23,24 **5**: 20,21,22,23,24		

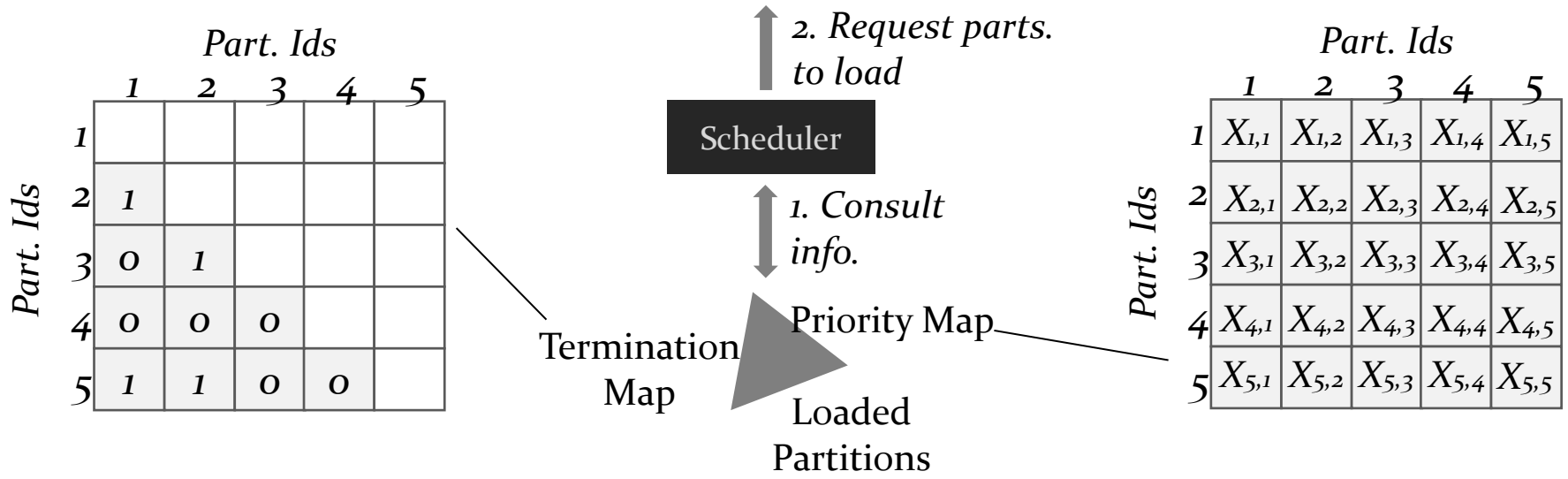
Graspan

Parallel EP-centric Computation Algorithm

	old	new	old U new	delta	old U new U delta
#1	<EMPTY>	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 2,3,10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24
#2	1: **2,3,**10,11,12 2: 4,13,15 3: 5,7,8,9,10 4: 5,17,18 5: 20,21,22,23,24	1: 4,5,7,8,9,13,15 2: 5,17,18 3: 20,21,22,23,24 4: 20,21,22,23,24 5:	1: 2,3,4,5,7,8,9,10,11,12,13,15 2: 4,5,13,15,17,18 3: 5,7,8,9,10,20,21,22,23,24 4: 5,17,18,20,21,22,23,24 5: 20,21,22,23,24	✔	✔
			. . .		
#n			<COMPUTATION O/P>	<EMPTY>	

Graspan

Scheduling



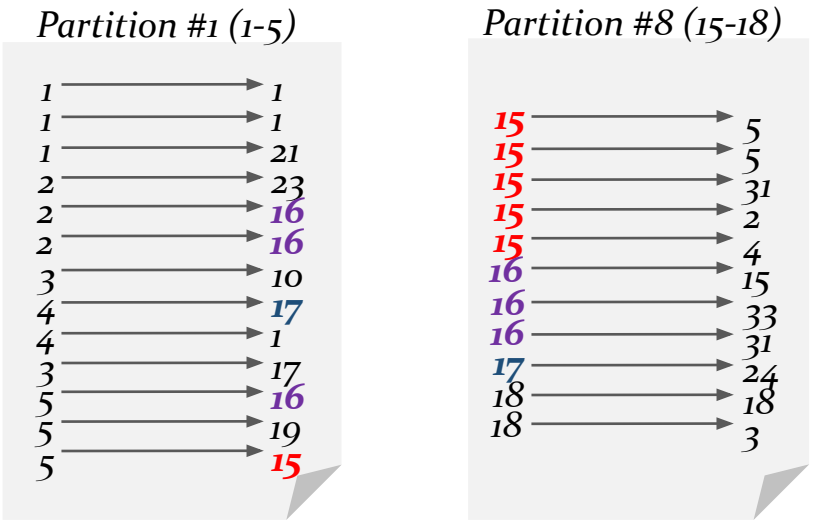
Our goal is to maximize the number of in-memory computations in a single load, and thus minimize the number of loads.

Different metrics can be used to guide us fulfill this objective.

Graspan

Scheduling

Edge Destination Count Metric



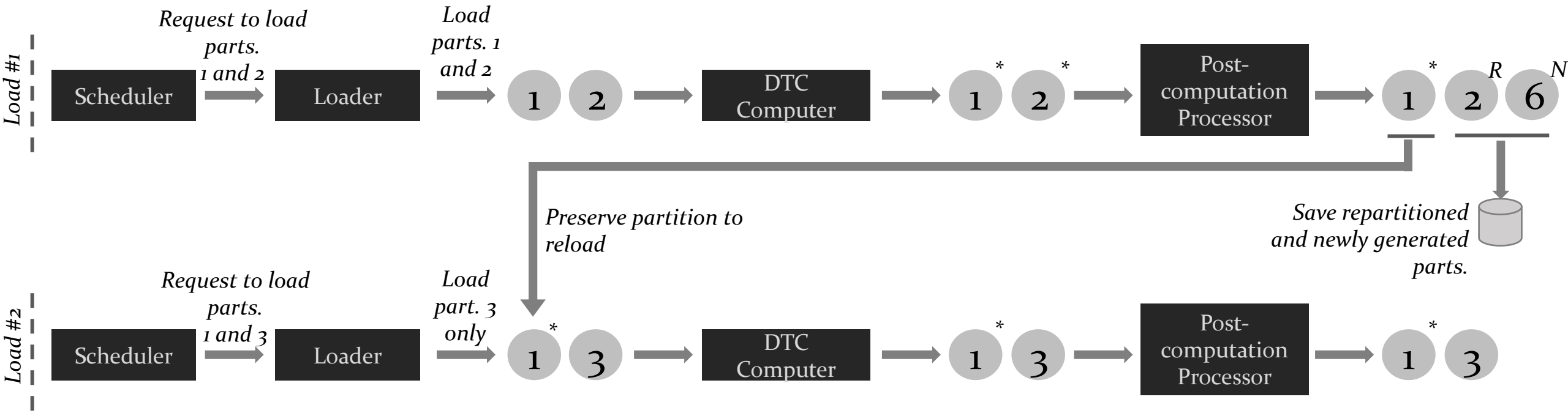
Aims to maximize the number of such matches.

For any partition pair *(Pi, Pj)*,
This metric can be edge based
(count # of edges in a partition that have source = to a target vertex in another partition),

or vertex based
(count # of unique source vertices in a partition = to a target vertex in another partition)

Graspan

Repartitioning



Partition Symbols:
* – Partition contains new edges.
R – Partition has been repartitioned.
N – Newly generated partition

Graspan

Evaluation

Setup

Programs analyzed

Program	Version	#LOC
Linux	4.4.0-rc5	16M
PostgreSQL	8.3.9	700K
Apache httpd	2.2.18	300K

Execution Environment

Dell Desktop Computer
Quad-Core 3.2GHZ Intel i5-4570 CPU
8GB Memory
1TB SSD
Linux 4.2.0

Graph Generation

Built graph generators based on LLVM Clang for
Pointer analysis (1.2K LOC) and
Dataflow analysis (800 LOC)
in C++

Static Analysis

Analyzing code without executing it

Useful for bug finding

Most existing techniques us

Null checker (Palix et al. ASPLOS 2011)

Finds whether pointers that can potentially return NULL values are checked before they are used.

Pattern matching static analysis techniques

Simple, easy to implement

Heuristic based and can miss de

The bug

If there is no such check.

Pattern used

See whether a check is used before using a pointer returned by a function that is known to always return NULL.

```
func1 ()
{
  int  * a= func2();
  //code works on a
  int b = *a;
}

int * func2 ()
{
 //code
  return x;
}
```

Such techniques have been found to miss bugs.

Graspan

Evaluation

Setup

Checkers Implemented

	Target Problems	How we aim to improve the checkers
Block	Deadlocks	Use a **pointer/alias analysis** to identify indirect invocations via function pointers of the blocking functions
Null	NULL pointer derefs	Use a **dataflow analysis** to identify functions where NULL can be propagated to their return variables
Range	Use user data as array index without checks	Use a **dataflow analysis** to identify indices transitively from user data as well
Lock/Intr	Double acquired locks and disabled interrupts not appropriately restored	Use a **pointer/alias analysis** to understand aliasing relationships among lock objects in different lock sites
Free	Use of a freed obj	Use a **pointer/alias analysis** to check if there is aliasing between objs freed and used afterwards
Size	Inconsistent sizes between an allocated obj and the type of the RHS var	Use a **pointer/alias analysis** to identify other vars that *point to* the obj with an inconsistent type

Graspan

Evaluation

Results

Graspan performance for pointer analysis

Program	Initial Graph Size		Final Graph Size		Preprocessing Time	Computation Time	#Rounds	#Rounds with Repartitioning
	#Edges	#Vertices	#Edges	#Vertices				
Linux	208.9M	43.3M	1.5B	43.3M	192 sec	4.1 hrs	563	21
PostgreSQL	177.3M	37.6M	1.1B	37.6M	127 secs	3.8 hrs	1,025	73
Apache httpd	4.5M	1M	155M	1M	4 secs	3.2 hrs	24,443	102

Graspan performance for dataflow analysis

Program	Initial Graph Size		Final Graph Size		Preprocessing Time	Computation Time	#Rounds	#Rounds with Repartitioning
	#Edges	#Vertices	#Edges	#Vertices				
Linux	49.5M	42.7M	170.4M	42.7M	178 secs	5.2 hrs	515	21
PostgreSQL	219.8M	169.9M	344.1M	169.9M	131 secs	2.5 hrs	245	3
Apache httpd	4.8M	3.5M	7.2M	3.5M	5 secs	4 mins	29	0

Pointer Analysis

Context Sensitivity

* Sridharan and Bodík
Refinement-based context-sensitive points-to analysis for Java. (PLDI 2006)

* Yan, Xu and Rountev
Demand-driven context-sensitive alias analysis for Java (ISSTA 2011)

* Zheng and Rugina
Demand-driven alias analysis for C (POPL 2008)

* Whaley and Lam
Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. (PLDI 2000)

* Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. POPL'96

* L. Anderson. Program analysis and specialization for the C programming language. University of Copenhagen. May 1994

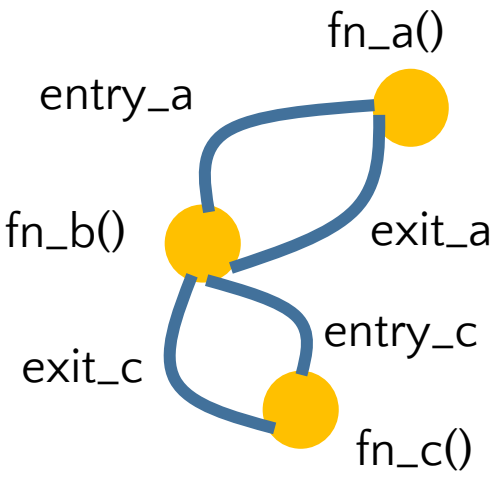
Pointer Analysis

Context Sensitivity

Most of these techniques model context sensitivity as a Context-Free-Language Reachability problem.

The language ensures that the entries and the exits of a function call are balanced.

A very simple example:



entry_i exit_i

Pointer Analysis

Demand Driven

* Sridharan and Bodík
Refinement-based context-sensitive points-to analysis for Java. (PLDI 2006)

* Zheng and Rugina
Demand-driven alias analysis for C (POPL 2008)

* Yan, Xu and Rountev
Demand-driven context-sensitive alias analysis for Java (ISSTA 2011)

Static Bug Finding

Annotation-Based Tools

* Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. (PLDI 2002)

* ESC/Java user`s manual. Technical note 2000-002, Compag Systems Research Center

Annotations inject knowledge into the analysis

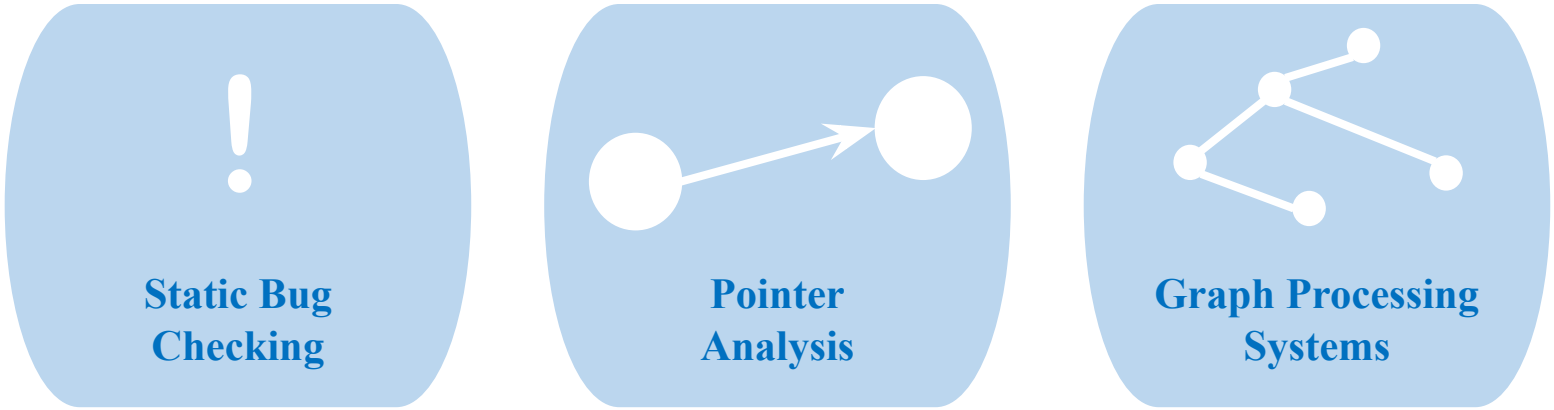
One Annotation per 50 lines of code[1]

Language-Based Tools

* **Coccinelle**
Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in linux: ten years later. SIGARCH Comput. Archit. News 39, 1 (March 2011), 305-318.

Use pattern matching.
Patterns are written in SmPL.

1 Cormac Flanagan and Stephen N. Freund. 2000. Type-based race detection for Java. InProceedings of the ACM SIGPLAN 2000 conference (PLDI '00)



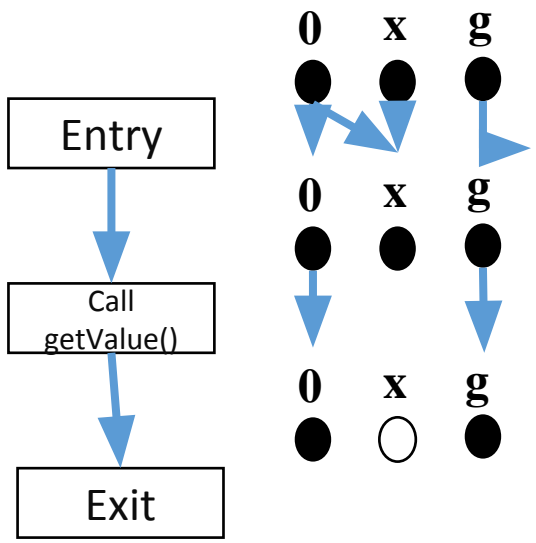
Related Fields

Thomas Reps et al.
Precise interprocedural dataflow analysis via graph reachability POPL 95

Transformed the the data-flow analysis problem to a graph reachability problem.

Enables the computation of dynamic transitive closure.

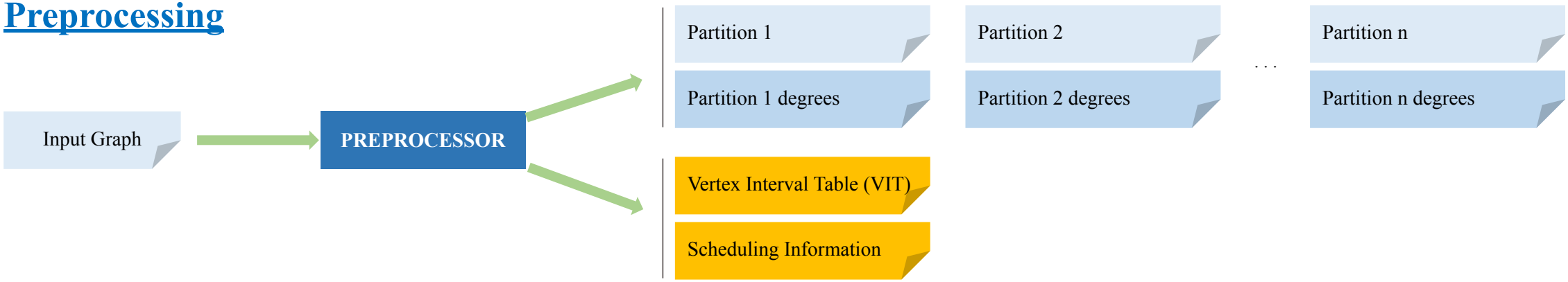
```
Main(){
int x, g;

X=getValue();

}
```



Graspan

Overall System Design

Preprocessing



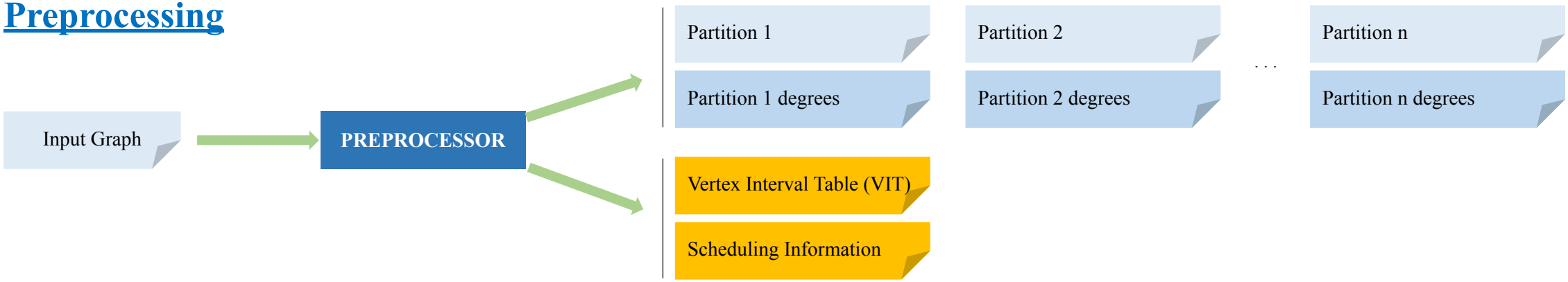
Properties of the partitions

1. All edges with the same source vertex id are in the same partition.
2. All partitions have roughly the same number of edges.
3. Source vertex ids in each partition are consecutive.

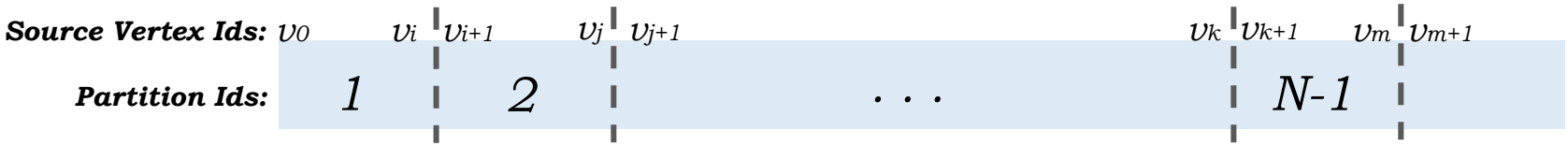
Graspan

Overall System Design

Preprocessing



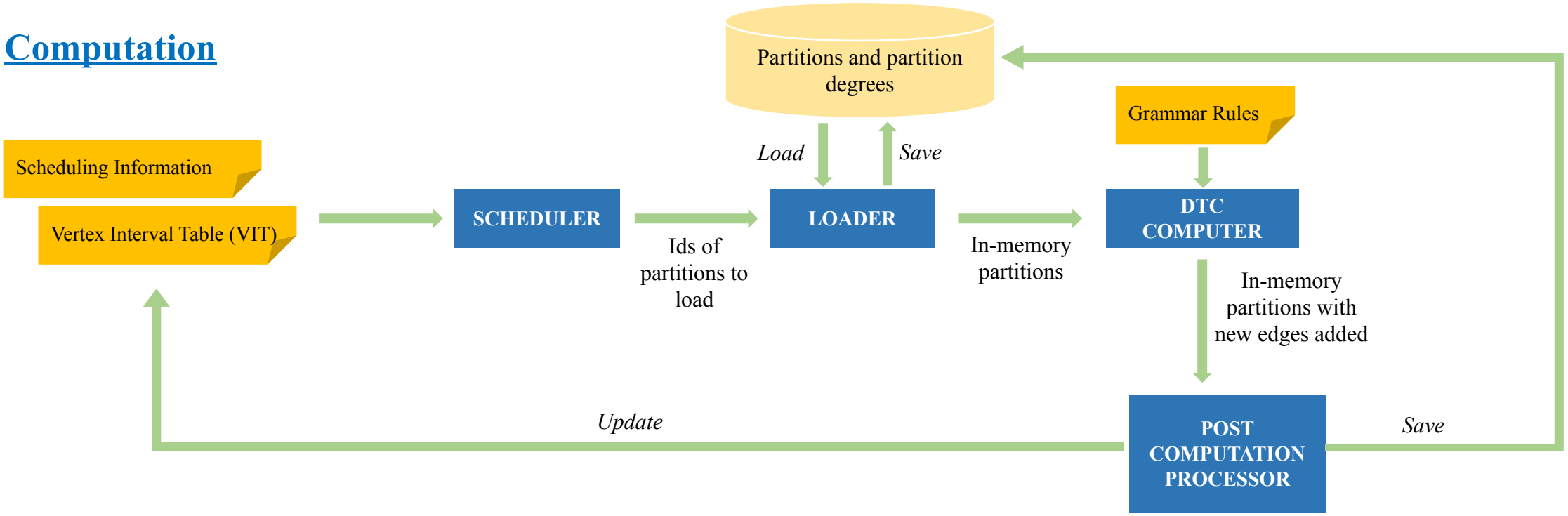
Vertex-Interval Table



Graspan

Overall System Design

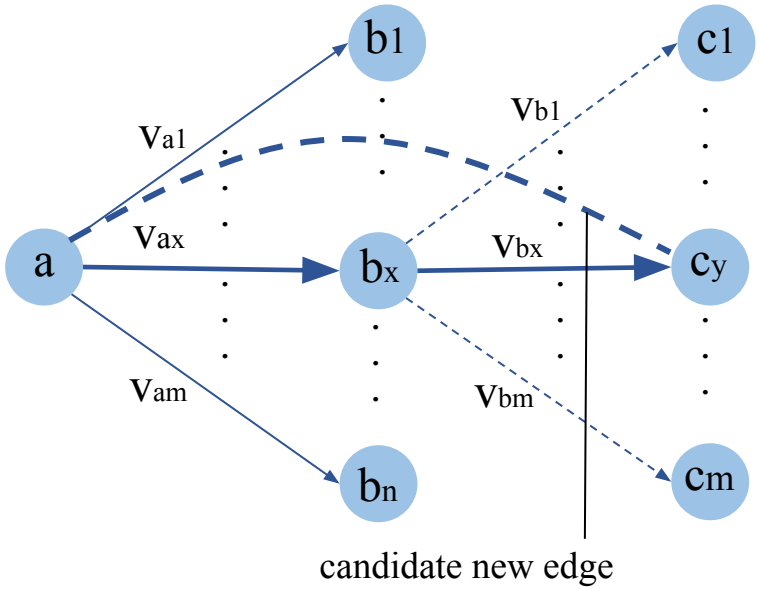
Computation



Graspan

Computation Model

Edge-Pair Centric Model



Graspan

Evaluation

Results

Bugs reported

	Baseline checker		Graspan analysis	
Checker	**#Reported Bugs**	**#False Positives**	**#Reported Bugs**	**#False Positives**
Block	0	0	0	0
Null	20	20	+108	23
Free	14	14	+4	4
Range	1	1	0	0
Lock	15	15	+3	3
Size	25	23	+11	11
Pnull	218	N/A	-218	0
UNTest	N/A	N/A	+1127	0

Graspan

Evaluation

Results

Graspan performance for pointer analysis

Program	Initial Graph Size
	#Edges
Linux	208.9M
PostgreSQL	177.3M
Apache httpd	4.5M

Graspan performance for dataflow analysis

Program	Initial Graph Size	Final Graph Size	Preprocessing Time	Computation Time	#Rounds	#Rounds with Repartitioning
	#Edges	#Edges				
Linux	49.5M	170.4M	178 secs	5.2 hrs	515	21
PostgreSQL	219.8M	344.1M	131 secs	2.5hrs	245	3
Apache httpd	4.8M	7.2M	5 secs	4 mins	29	0

Interprocedural Static Analysis

Uses calling relationships among procedures.

Enables to obtain more precise analysis information.

Dataflow Analysis – helps in finding transitive flow info.

Pointer Analysis – helps in finding aliases.

Interprocedural Static Analysis

```
int * function1() {

  int * ptr1;
  if (<a condition>) {
    ptr1 = fn_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
return ptr1;
}
```

```
function2() {
   int * ptr2 = function1();

   int c = *ptr2;
}
```

Interprocedural Static Analysis

```
int * function1() {

  int * ptr1;
  if (<a condition>) {
    ptr1 = fn_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
return ptr1;
}
```

```
function2() {
  int * ptr2 = function1();

  int c = *ptr2;
}
```

NEED NULL CHECK

Interprocedural Static Analysis

```
int * function1() {

  int * ptr1;
  if (<a condition>) {
    ptr1 = fn_ret_null();
  }
  else {
    ptr1 = fnA();
  }
  if (ptr1!=NULL) {
    int b = *ptr1;
  }
return ptr1;
}
```

```
function2() {
   int * ptr2 = function1();

   int c = *ptr2;
}
```

Interprocedural Static Analysis

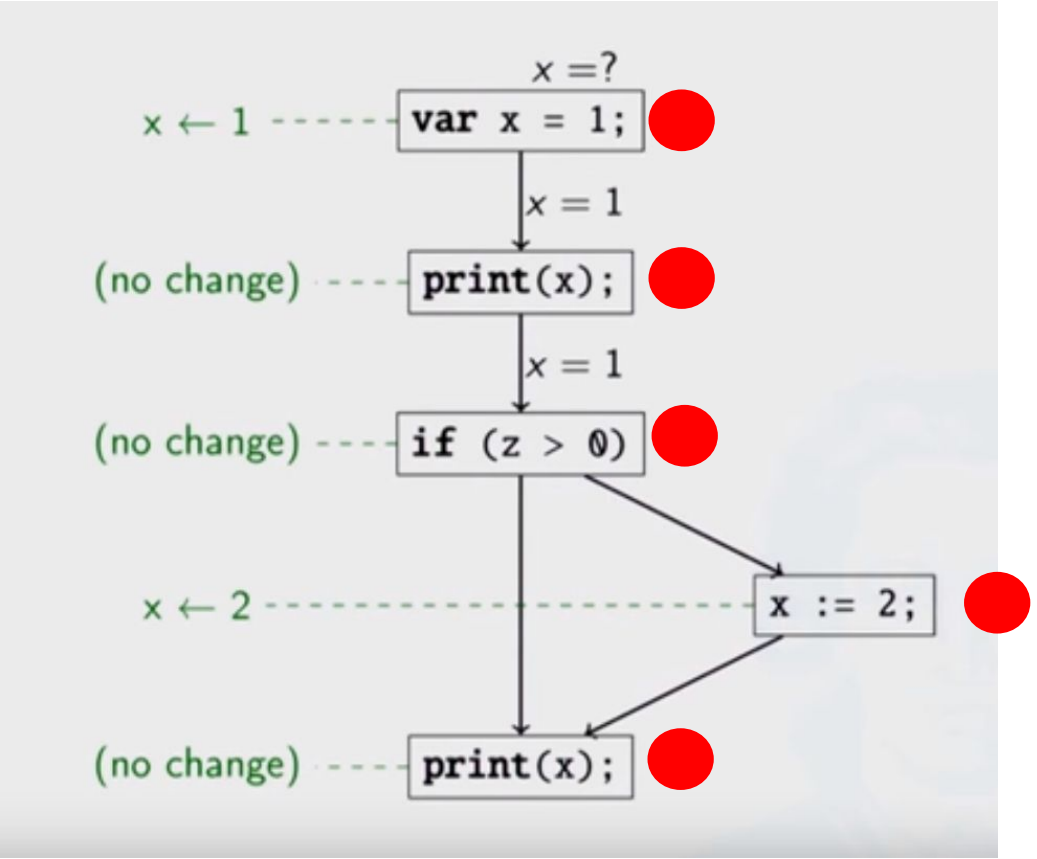
Uses calling relationships among procedures.

Enables to obtain more precise analysis information.

Dataflow Analysis – helps in finding transitive flow info.

Pointer Analysis – helps in finding aliases.

Dataflow Analysis



Transfer functions

Pointer Analysis

A pointer analysis computes, for each pointer variable, a set of heap objects that can flow to the variable.

x = **new** A(); → o

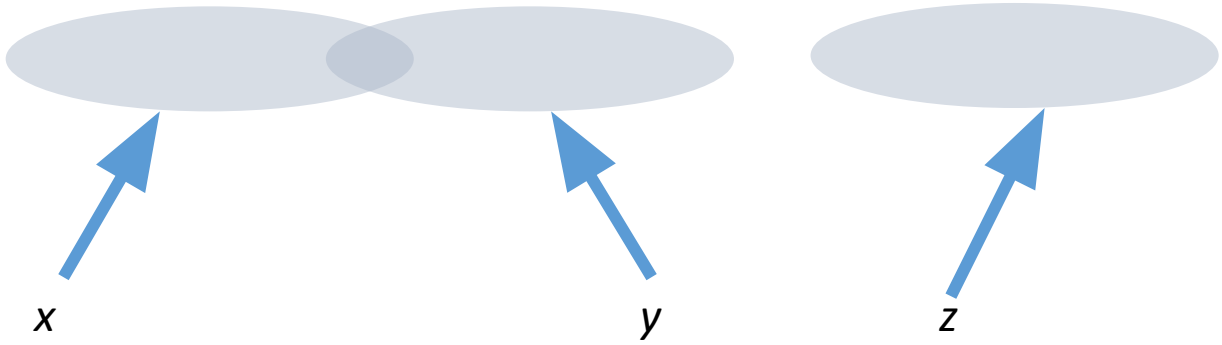
Allocates memory in the heap space.
This is an *allocation site*.

For each allocation site, we create an *abstract location* in the heap, where we assume the object will be created.

Due to the assignment,
"*x* points to *o*" and hence *o* belongs to points-to set of *x*, *pt(x)*

Pointer Analysis

A pointer analysis computes, for each pointer variable,
a set of heap objects that can flow to the variable.



x and y may alias

Pointer Analysis

Different properties

Flow Sensitivity

Field Sensitivity

Context Sensitivity

Pointer Analysis

Different properties

Flow Sensitivity

Field Sensitivity

Context Sensitivity

```
struct  {
  int f1;
  int f2;
} x, y;
```

x.f1 x.f2 y.f1 y.f2	x.* y.*	*.f1 *.f2
Field sensitive	Field insensitive	Field based

Pointer Analysis

Different properties

Flow Sensitivity

Field Sensitivity

Context Sensitivity

Most significant
According to study by
Hind, Pointer Analysis: Haven't We Solved This Problem Yet?
PASTE 2001

Pointer Analysis

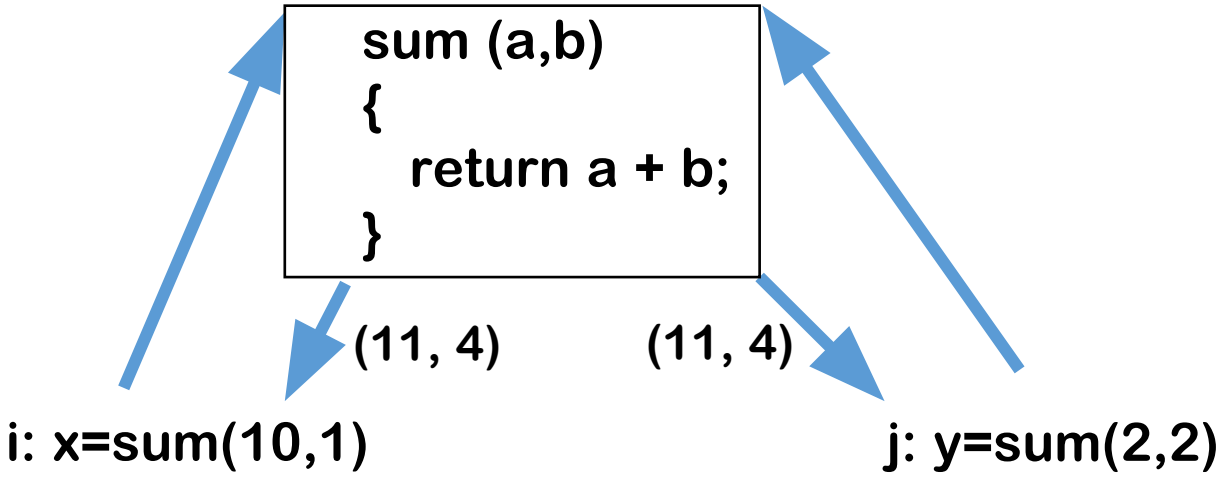
Different properties

Flow Sensitivity

Field Sensitivity

Context Sensitivity

Distinguishes calling contexts.



Adapted from Radu Rugina, CS711

Pointer Analysis

Context Insensitive Pointer Analysis

* Bjarne Steensgaard. Points-to Analysis in Almost Linear Time. POPL'96

* L. Anderson. Program analysis and specialization for the C programming language. University of Copenhagen. May 1994

* Manuvir Das. 2000. Unification-based pointer analysis with directional assignments. In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI '00).

* Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to analysis using BDDs. SIGPLAN Not. 38, 5 (May 2003), 103-114.

Context Sensitive Pointer Analysis

* Sridharan and Bodík
Refinement-based context-sensitive points-to analysis for Java. (PLDI 2006)

* Yan, Xu and Rountev
Demand-driven context-sensitive alias analysis for Java (ISSTA 2011)

* Zheng and Rugina
Demand-driven alias analysis for C (POPL 2008)

* Whaley and Lam
Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. (PLDI 2000)

* Robert P. Wilson and Monica S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs.
In Proceedings of the ACM SIGPLAN (PLDI conference on '95).

* Samuel Z. Guyer and Calvin Lin. 2003. Client-driven pointer analysis. Radhia Cousot (Ed.). Springer-Verlag, Berlin, Heidelberg, 214-236.

Context Insensitive Pointer Analysis

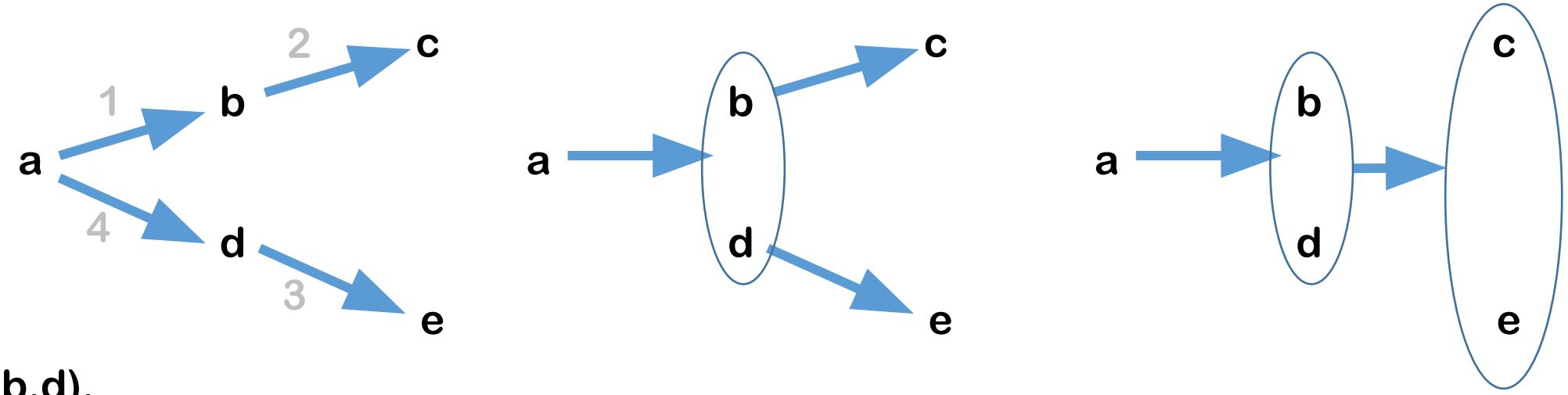
Bjarne Steensgaard
Points-to Analysis in Almost Linear Time. POPL` 96

* Unification Based pointer analysis algorithm, that generates points-to sets of variables in a program.

p = q

PointsTo(p) = PointsTo(q)

1 a = &b
2 b = &c
3 d = &e
4 a = &d



PointsTo(a)=(b,d),
PointsTo(b,d)=(c,e)

Context Insensitive Pointer Analysis

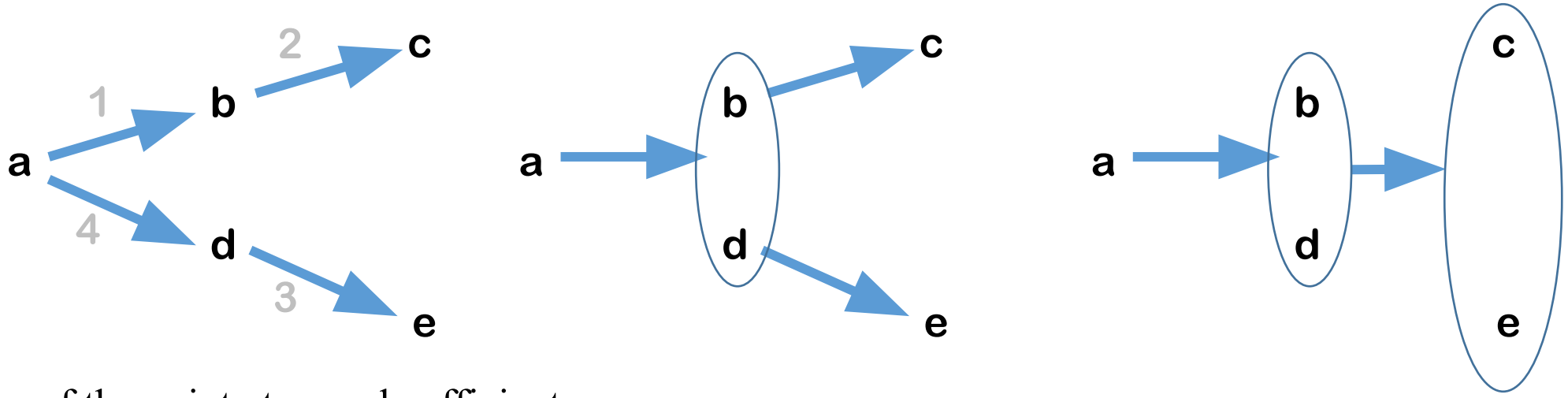
Bjarne Steensgaard
Points-to Analysis in Almost Linear Time. POPL` 96

* Unification Based pointer analysis algorithm, that generates points-to sets of variables in a program.

p = q

PointsTo(p) = PointsTo(q)

1 a = &b
2 b = &c
3 d = &e
4 a = &d



* Reduces the size of the points-to graph, efficient.
* Imprecise

Context Insensitive Pointer Analysis

L. Anderson.
Program analysis and specialization for the C programming language. University of Copenhagen. May 1994

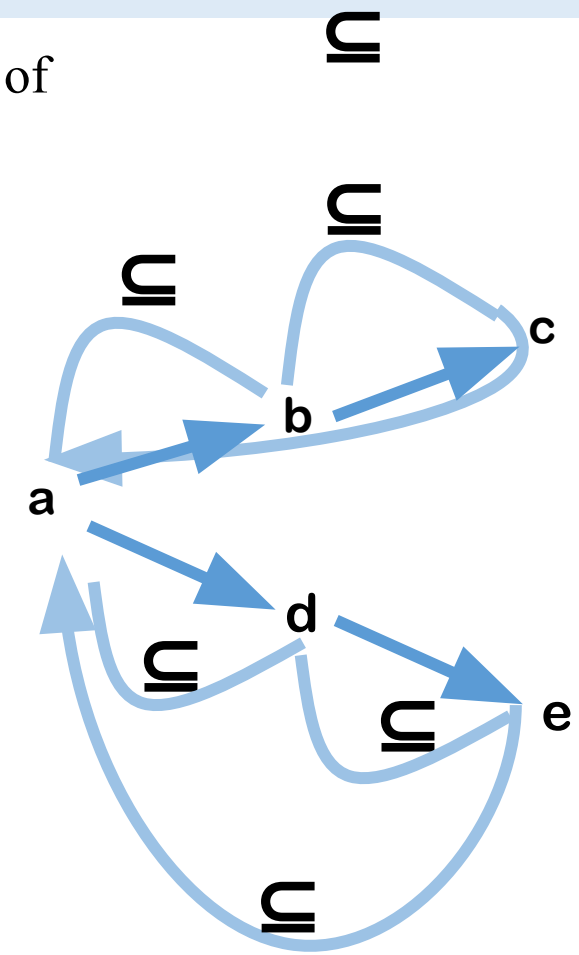
* Inclusion Based pointer analysis algorithm, that generates points-to sets of variables in a program.

p = q

PointsTo(q) ⊆ PointsTo(p)

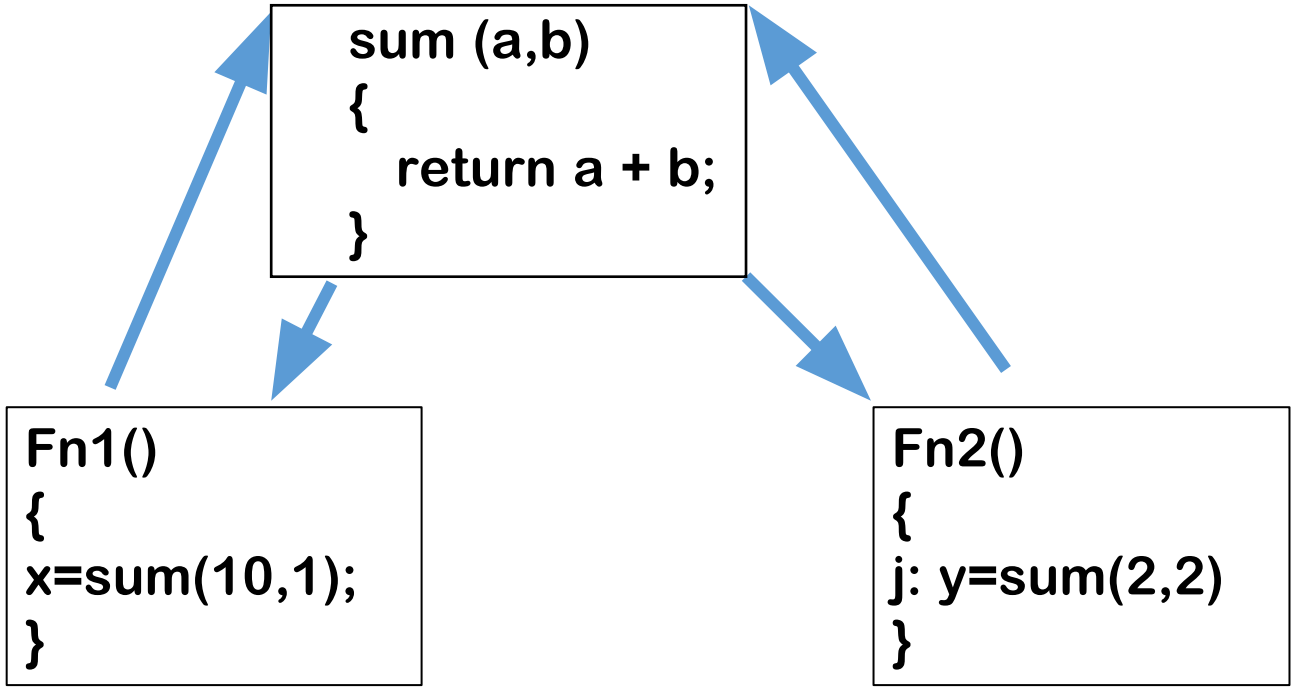
* More precise than Steensgard.

```
1 a = &b
2 b = &c
3 d = &e
4 a = &d
```

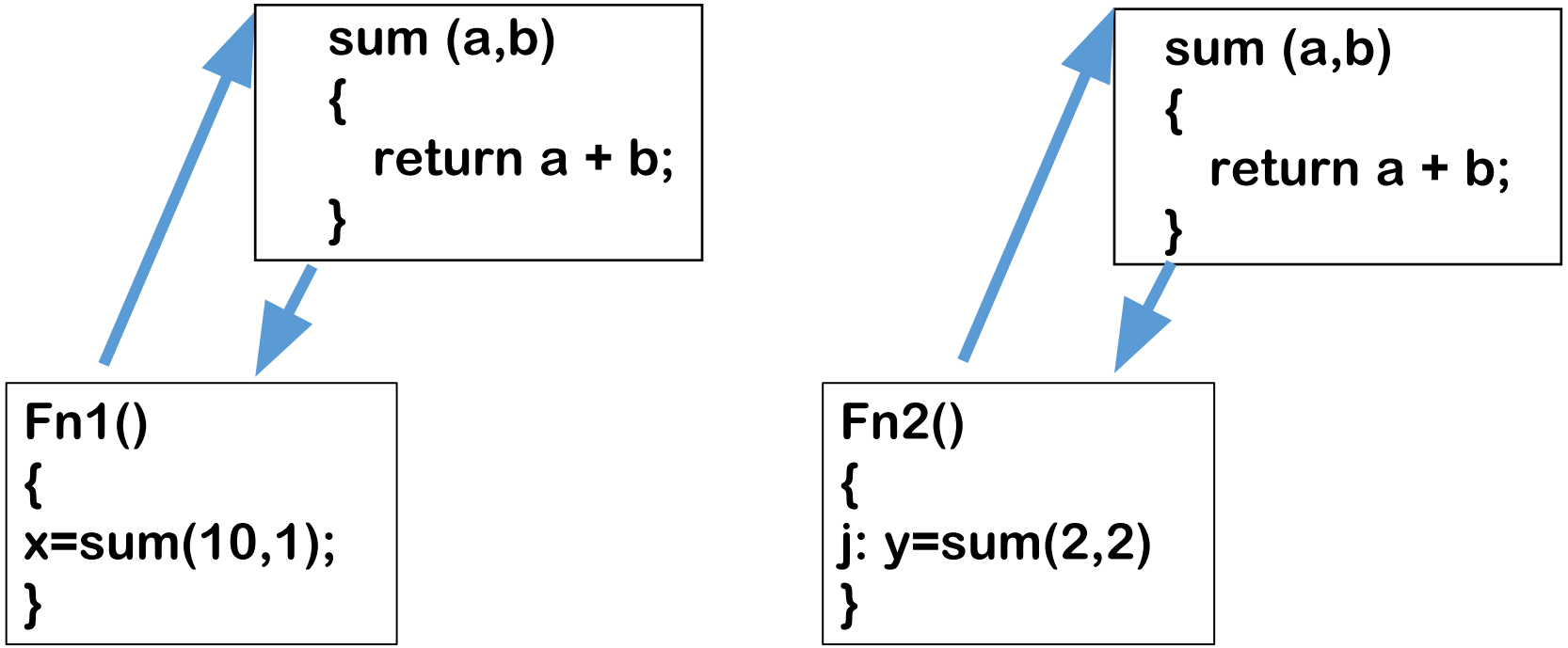


* Points-to graph is larger, and it takes longer to finish.

Context Sensitive Pointer Analysis

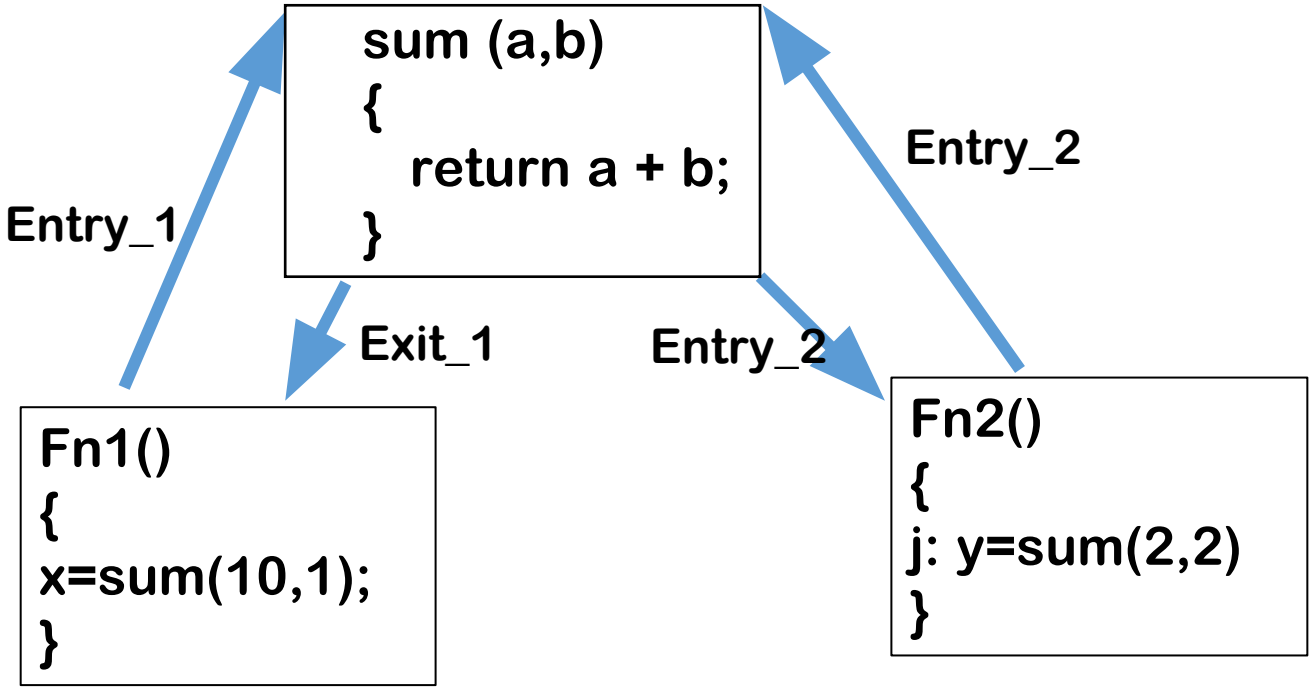

sum (a,b)
{
return a + b;
}
Fn1()
{
x=sum(10,1);
}
Fn2()
{
j: y=sum(2,2)
}

Cloning



* Whaley and Lam
Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. (PLDI 2000)

Label matching



Entry_1, Exit_1

Entry_1, Exit_2

Entry_2, Exit_1

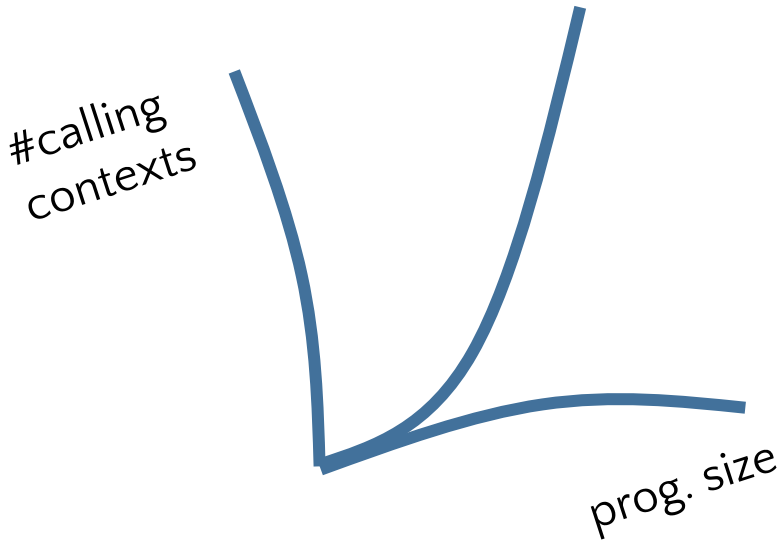
Entry_2, Exit_2

* Sridharan and Bodík
Refinement-based context-sensitive points-to analysis for Java. (PLDI 2006)

* Yan, Xu and Rountev
Demand-driven context-sensitive alias analysis for Java (ISSTA 2011)

* Zheng and Rugina
Demand-driven alias analysis for C (POPL 2008)

Exponential Contexts!



Demand Driven

Zheng and Rugina
Demand-driven alias analysis for C (POPL 2008)

Answers alias queries

They accurately answered 96% of their queries in 0.5s
Used the SPEC Benchmark 2000

The largest graph they analyzed had 800,000 edges.

Client Driven

Sridharan and Bodík
Refinement-based context-sensitive points-to analysis
for Java. (PLDI 2006)

Budget

**Provide a conservative,
heuristic based answer
upon termination**