A Probabilistic Pointer Analysis for Speculative Optimization

Jeff DaSilva

Greg Steffan

Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada

Oct 17th, 2005

Pointers Impede Optimization

- Many optimizations come to a halt when they encounter an ambiguous pointer

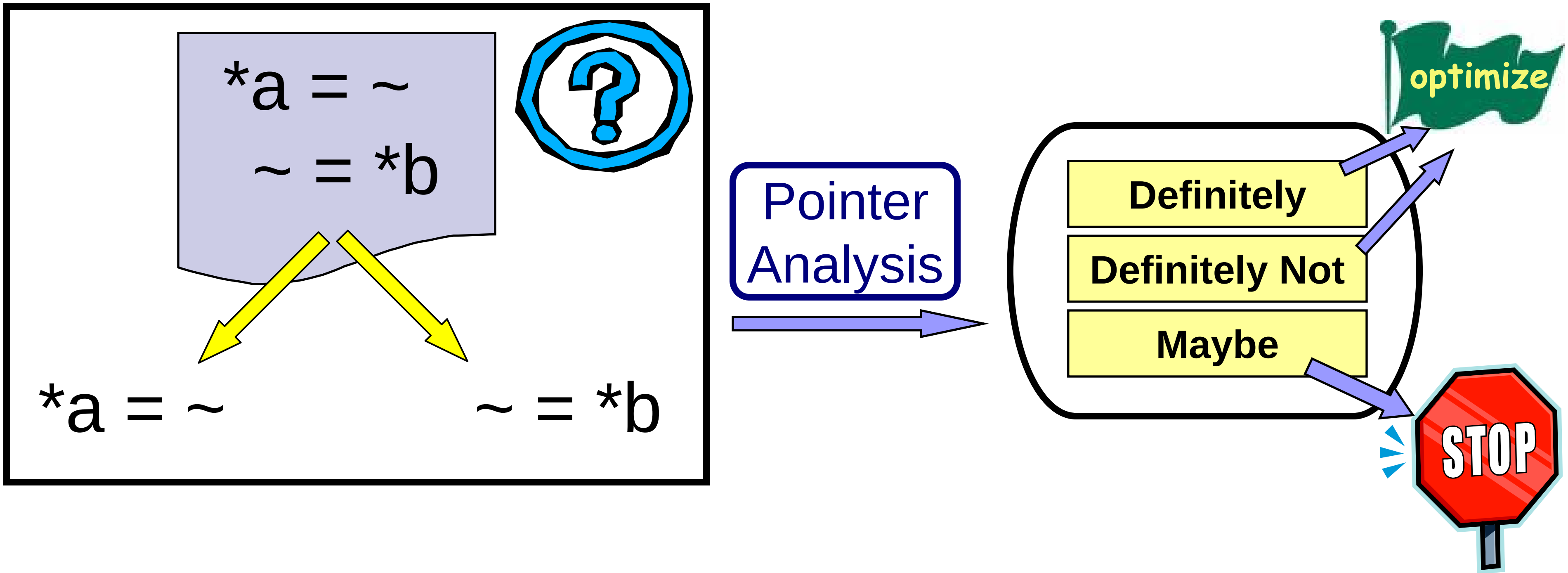
```
foo(int *a) {
 …
 while(…)
 {
   x = *a;
   …
 }
}
```

Loop Invariant Code Motion

Parallelize

☞**Pointer Analysis is Important**

Pointer Analysis



- Do pointers ***a*** and ***b*** point to the same location?
  - Do this for every pair of pointers at every program point

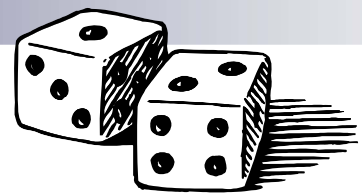
Pointer Analysis is Difficult

- Pointer analysis is a difficult problem
  - **scalable** and **overly conservative**

    or
  - **fails-to-scale** and **accurate**

- Ambiguous pointers will persist
  - even when using the most **accurate** of algorithms
  - **Maybe** output is often unavoidable
- What can be done with **Maybe** ?

Lets Speculate



- Compilers make **conservative** assumptions
  - They must always preserve program correctness

"It's easier to apologize than ask for permission."

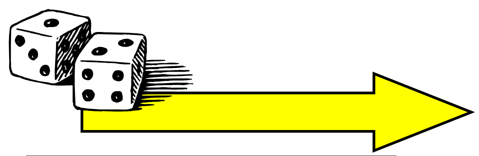
Author: Anonymous

Implement a potentially **unsafe** optimization

Verify *and* ***Recover*** *if necessary*

Speculation applied to Pointers

```
int *a, x;
...
while(...)
{
   x = *a;
   ...
}
```

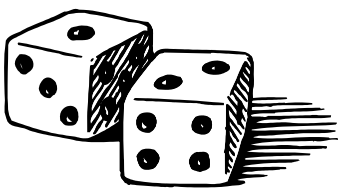


a is probably loop invariant

```
int *a, x, tmp;
...
tmp = *a;
while(...)
{
   x = tmp;
   ...
}
<verify, recover?>
```

Data Speculative Optimizations

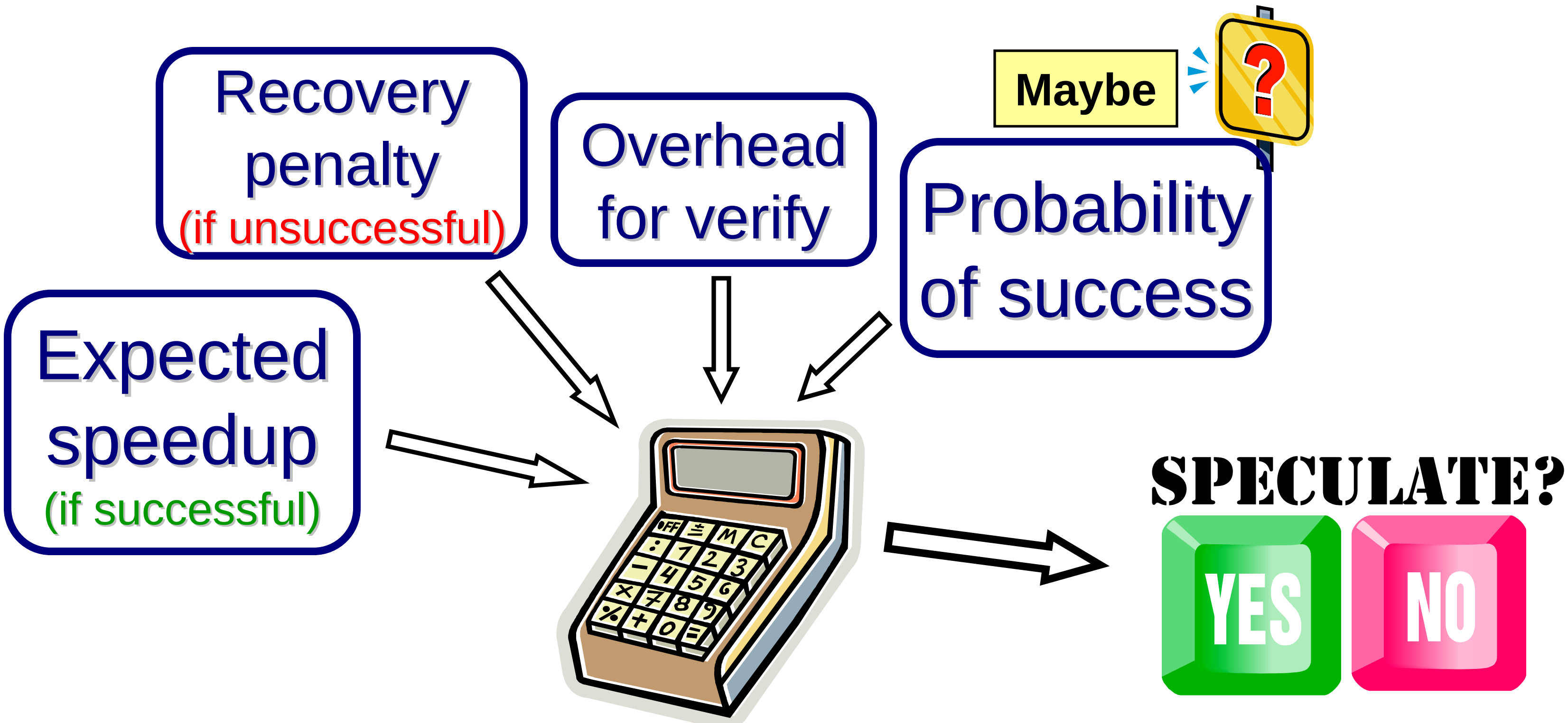
- The EPIC Instruction set
  - Explicit support for speculative load/store instructions (eg. Itanium)
- Speculative compiler transformations
  - Dead store elimination, redundancy elimination, copy propagation, strength reduction, register promotion
- Thread-level speculation (TLS)
  - Hardware support for tracking speculative parallel threads
- Transactional programming
  - Rollback support for aborted transactions



☞ **When to speculate? Techniques rely on profiling**

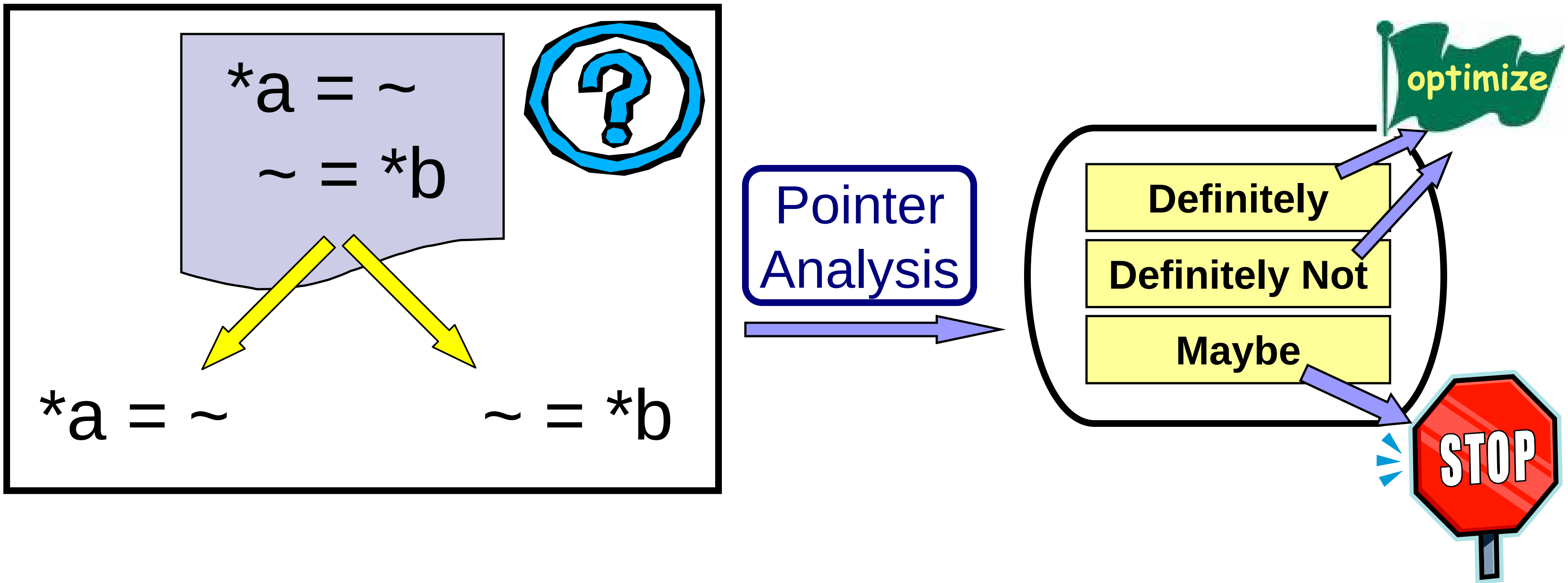
Quantitative **Maybe** Output Required

- Estimate the potential benefit for speculating:



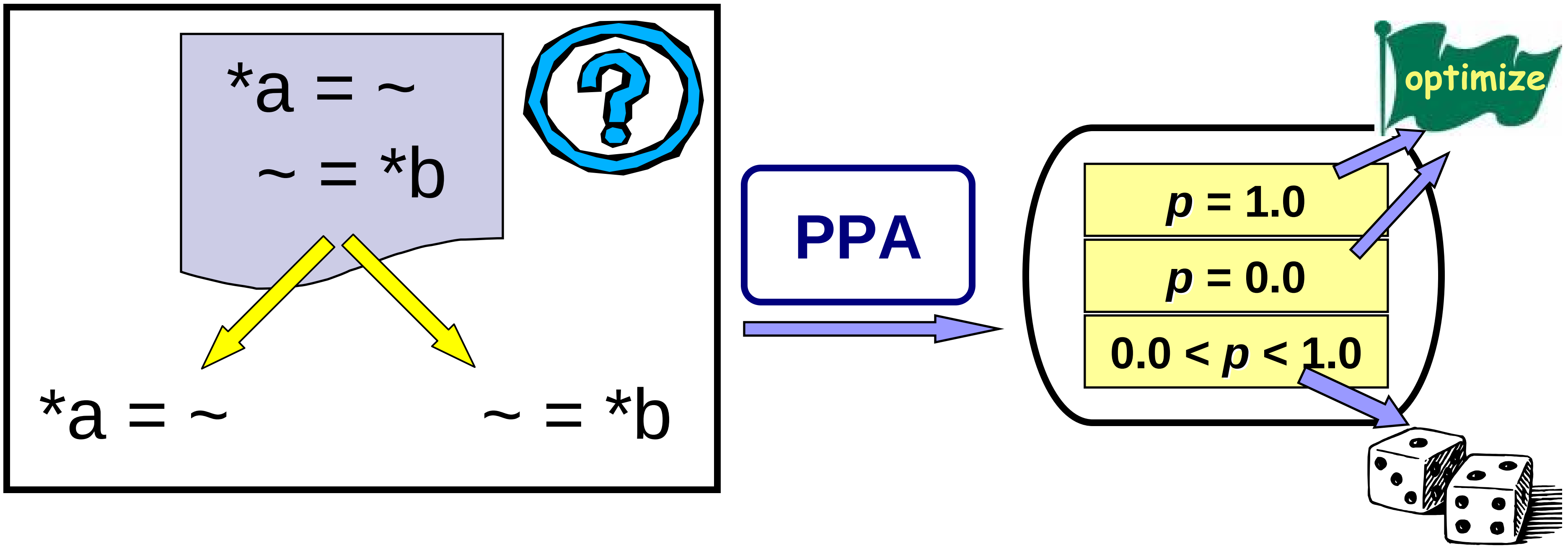
☞ **Probabilistic** **Maybe** **output needed**

Conventional Pointer Analysis



- Do pointers ***a*** and ***b*** point to the same location?
  - Do this for every pair of pointers at every program point

Probabilistic Pointer Analysis **(PPA)**



- With what probability ***p***, do pointers ***a*** and ***b*** point to the same location?
  - Do this for every pair of pointers at every program point

PPA Research Objectives

- Accurate points-to probability information
  - at every static pointer dereference
- Scalable analysis
  - Goal: The entire SPEC integer benchmark suite
- Understand scalability/accuracy tradeoff
  - through flexible static memory model
- Improve our understanding of programs

Algorithm Design Choices

- **Fixed**
  - Bottom Up / Top Down Approach
  - **Linear** transfer functions (for scalability)
  - One-level **context** and **flow** sensitive
- **Flexible**
  - Edge profiling (or static prediction)
  - Safe (or unsafe)
  - Field sensitive (or field insensitive)

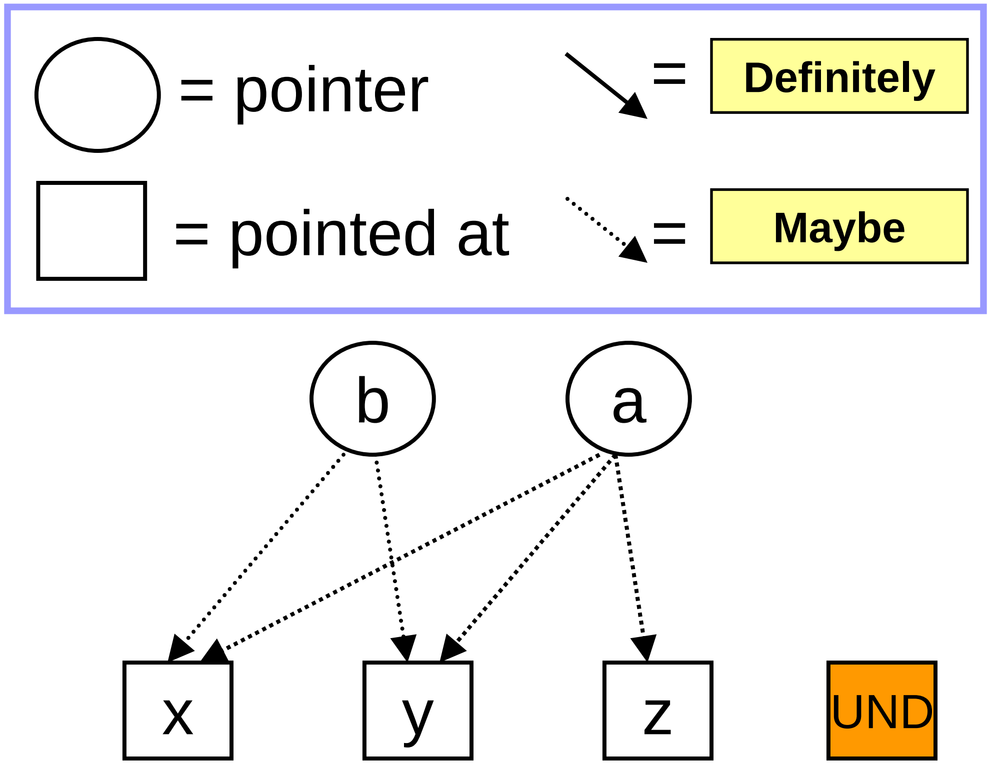
Traditional Points-To Graph

```
int x, y, z, *b = &x;
void foo(int *a) {

  if(…)
    b = &y;

  if(…)
    a = &z;
  else(…)
    a = b;

  while(…) {
    x = *a;
    …
  }
}
```



☞ **Results are inconclusive**

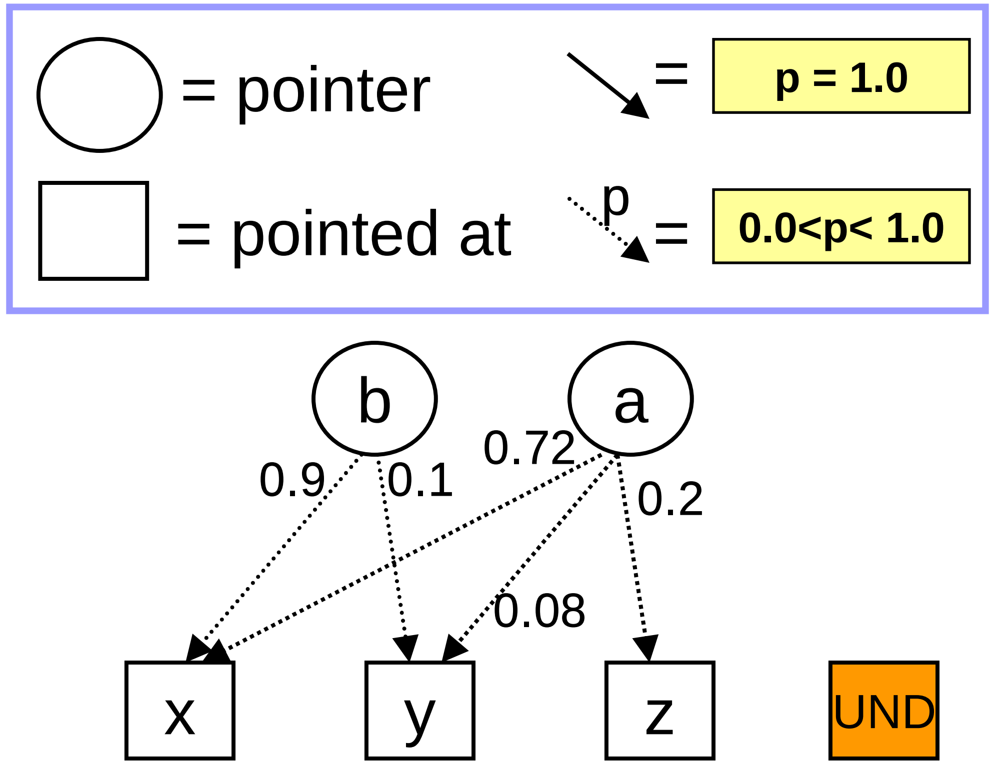
Probabilistic Points-To Graph

```
int x, y, z, *b = &x;
void foo(int *a) {

   if(…)⇨0.1 taken(edge profile)
      b = &y;

   if(…)⇨0.2 taken(edge profile)
      a = &z;
   else(…)
      a = b;
→

   while(…) {
      x = *a;
      …
   }
}
```



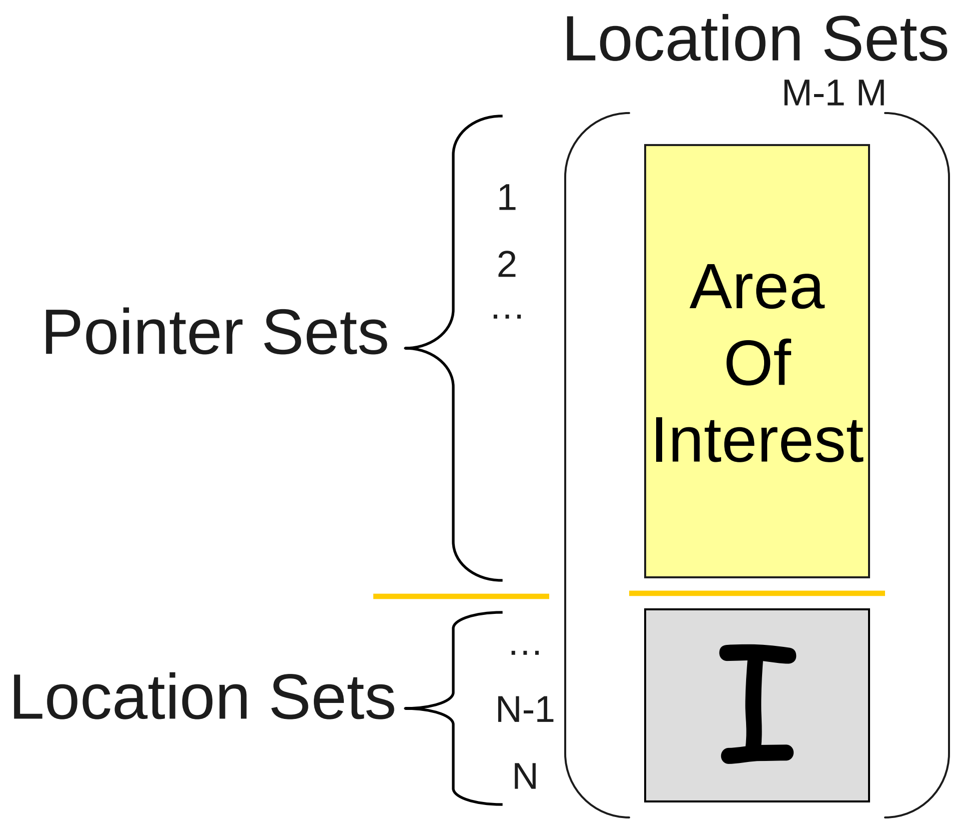
☞ **Results provide more information**

Linear
One -
Level
Interprocedural
Probabilistic
Pointer Analysis

LOLLIPOP

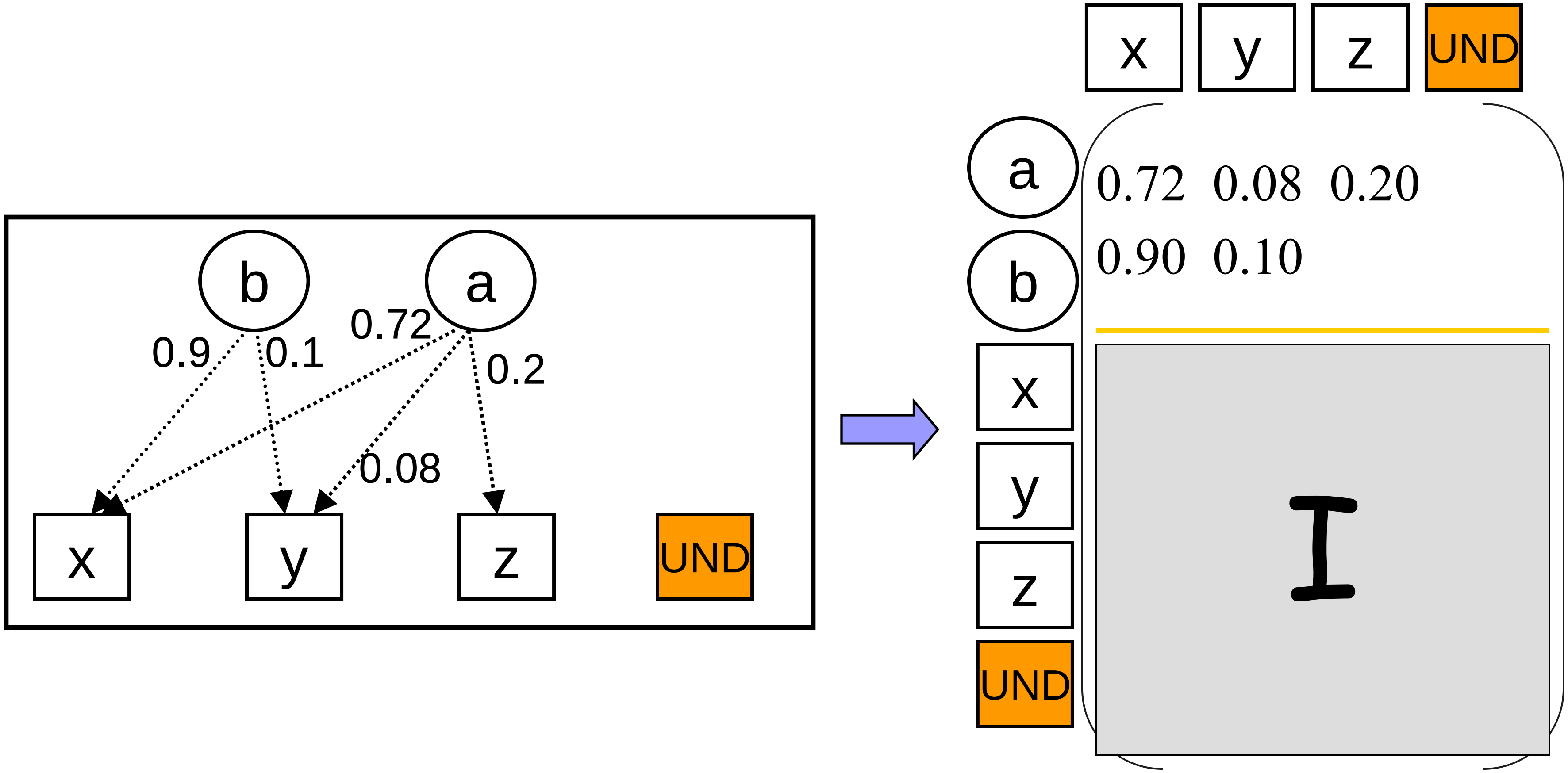
Our PPA Algorithm

Points-To Matrix



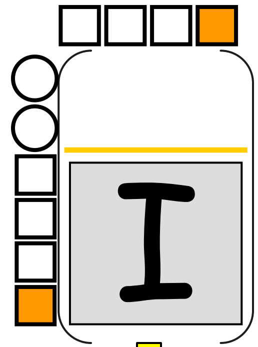
☞ **All matrix rows sum to 1.0**

Points-To Matrix Example



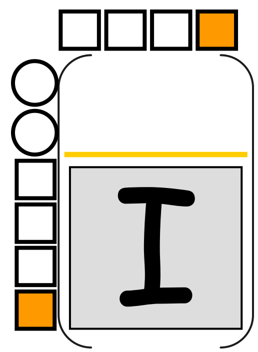
Solving for a Points-To Matrix

Points-To
Matrix In



Any Instruction

Points-To
Matrix Out

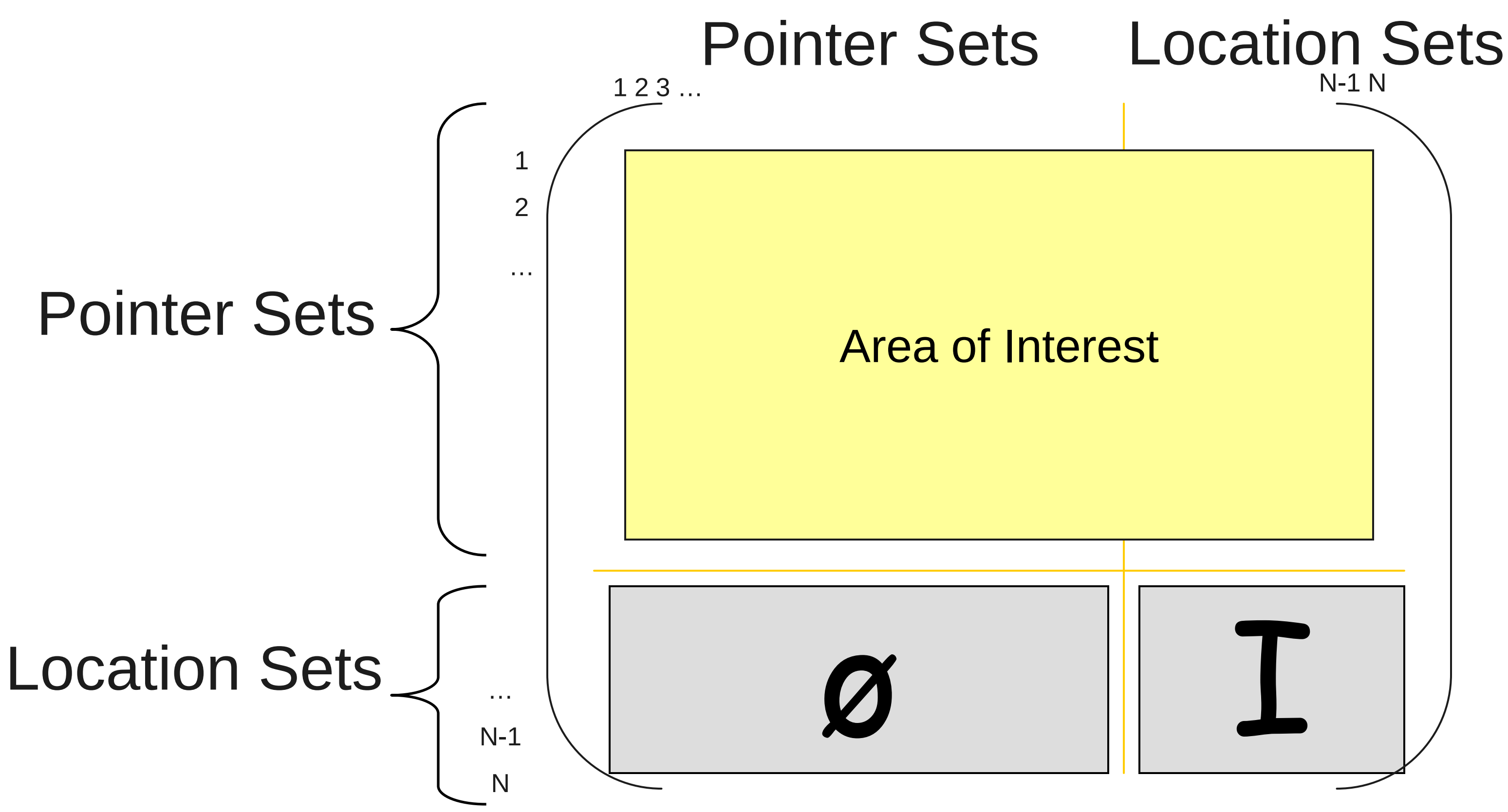


The Fundamental PPA Equation

$$\begin{pmatrix}\text{Points-To} \\ \text{Matrix Out}\end{pmatrix} = \begin{pmatrix}\text{Transformation} \\ \text{Matrix}\end{pmatrix}\begin{pmatrix}\text{Points-To} \\ \text{Matrix In}\end{pmatrix}$$

☞ **This can be applied to any instruction (incl. function calls)**

Transformation Matrix



☞ **All matrix rows sum to 1.0**

Transformation Matrix Example

S1: a = &z;

$[T_{S1}] =$

	a	b	x	y	z	UND
a					1.0	
b		1.0				
x			1.0			
y				1.0		
z					1.0	
UND						1.0

Example - The PPA Equation

$$\left(PT_{out}\right) = \left(T_{S1} \right)\left(PT_{in}\right)$$

S1: a = &z;

$$\left(PT_{out}\right) = \left(\begin{array}{cc|cccc} & & & & 1.0 & \\ & 1.0 & & & & \\ \hline & & 1.0 & & & \\ & & & 1.0 & & \\ & & & & 1.0 & \\ & & & & & 1.0 \end{array}\right)\left(\begin{array}{cccc} 0.72 & 0.08 & 0.20 & \\ 0.90 & 0.10 & & \\ \hline 1.0 & & & \\ & 1.0 & & \\ & & 1.0 & \\ & & & 1.0 \end{array}\right)$$

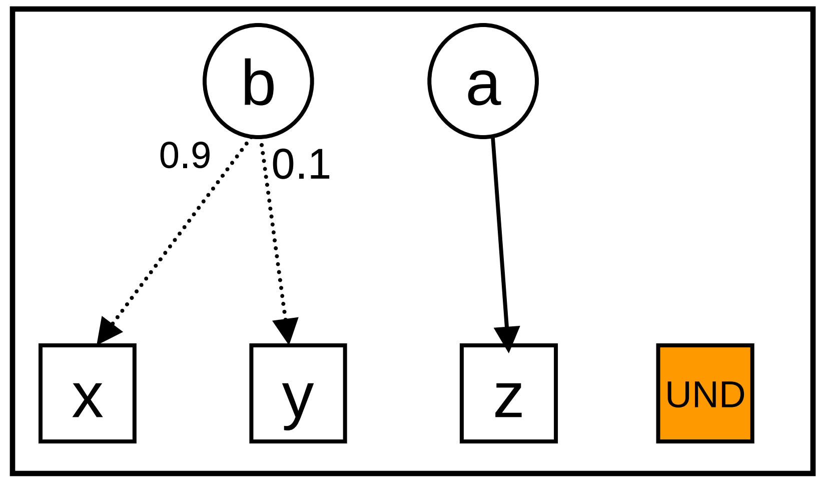
Example - The PPA Equation

$$PT_{out} = T_{S1} \, PT_{in}$$

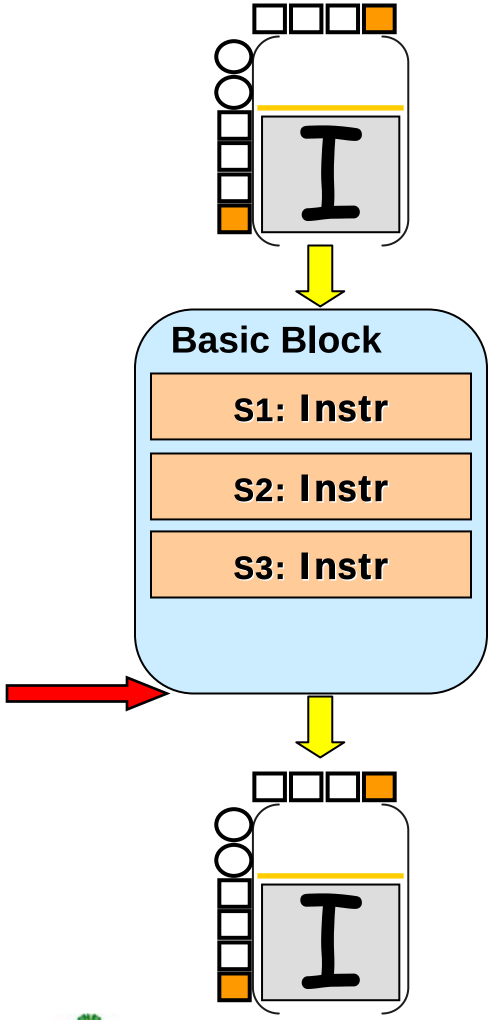
S1: a = &z;

$$PT_{out} =$$

	x	y	z	UND
a			1.0	
b	0.90	0.10		
x	1.0			
y		1.0		
z			1.0	
UND				1.0



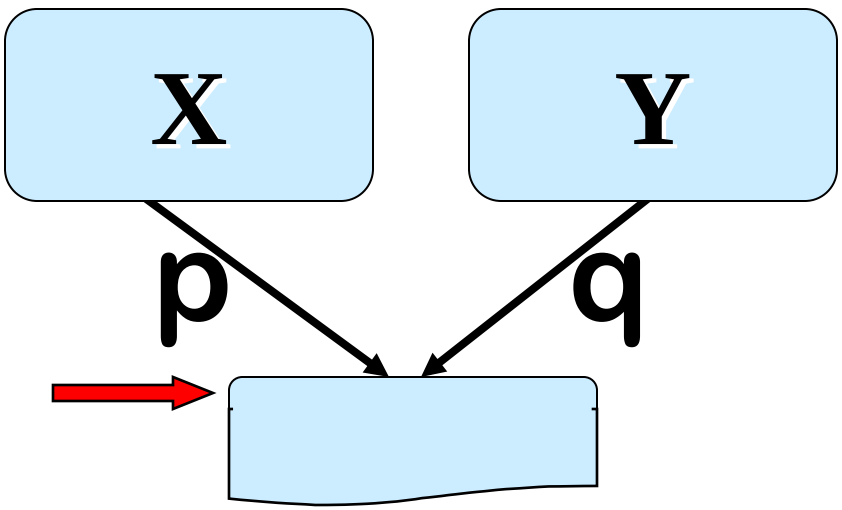
Combining Transformation Matrices



$$\left[PT_{out}\right] = \left[T_{S3}\right]\left[T_{S1}\right]\left[T_{S1}\right]\left[PT_{in}\right]$$

$$\left[PT_{out}\right] = \left[T_{BB}\right]\left[PT_{in}\right]$$

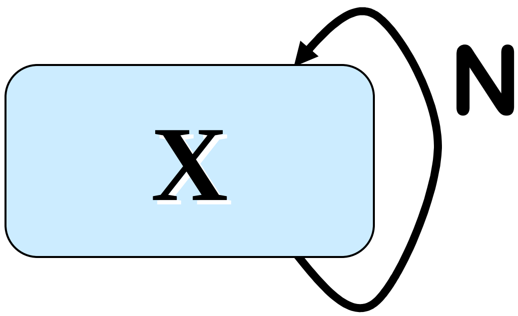
Control flow - if/else



$$p + q = 1.0$$

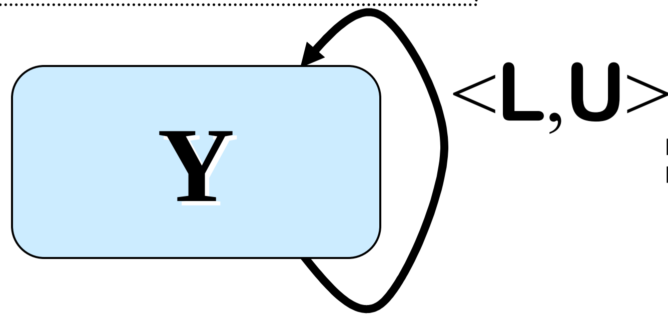
$$= p\left(T_X \right) + q\left(T_Y \right)$$

Control flow - loops



$$= \left(T_X \right)^N$$

<L,U> ⇨ <min,max>



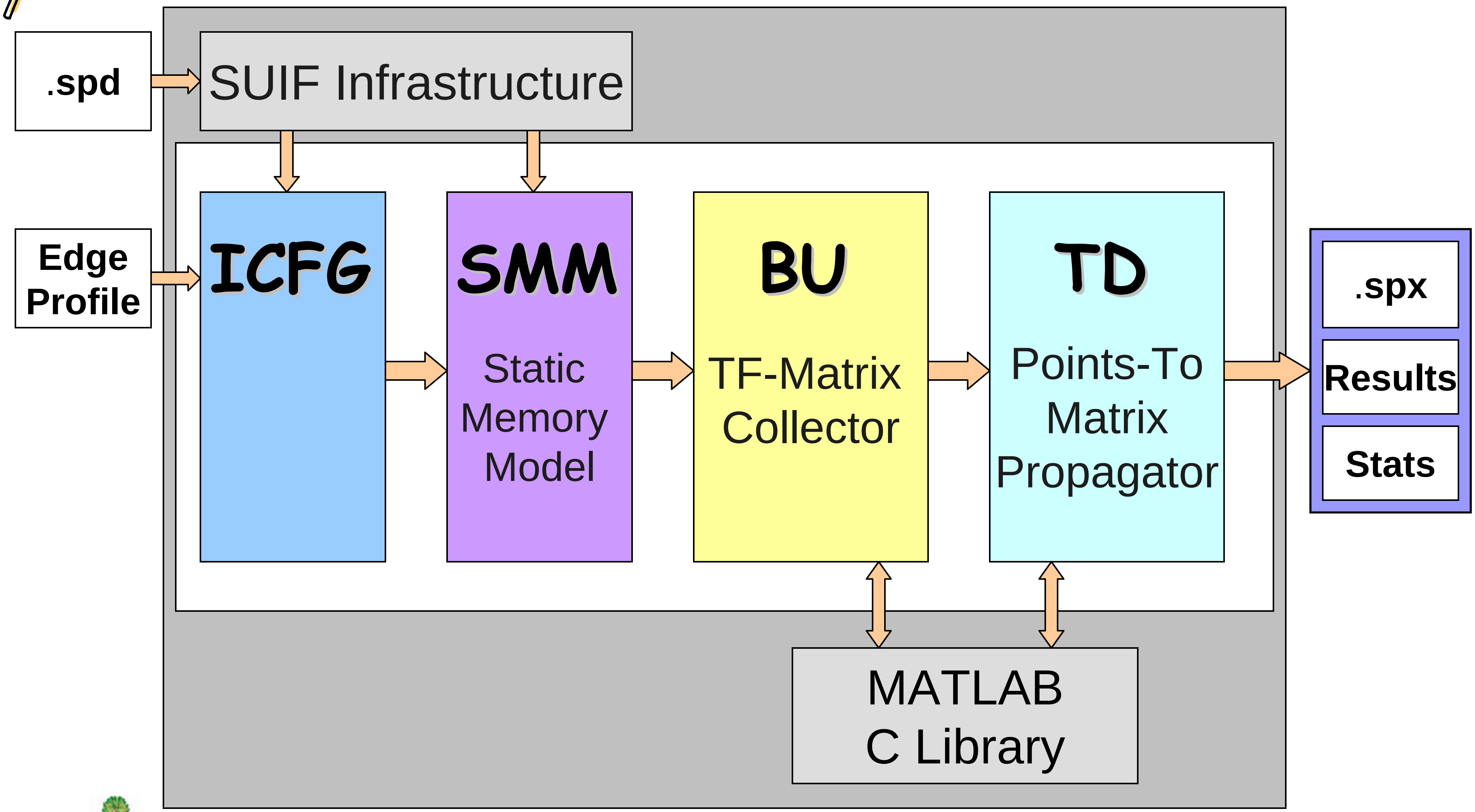
$$= \frac{1}{U-L+1} \sum_{i=L}^{U} \left(T_Y \right)^i$$

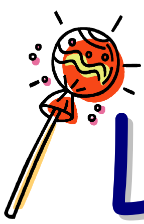
☞ **Both operations can be implemented efficiently**

Safe vs. Unsafe Pointer Assignment Instructions

		Safe?
x = &y	Address-of Assignment	☑
x = y	Copy Assignment	☑
x = *y	Load Assignment	☑ ☒
*x = y	Store Assignment	☑ ☒

LOLLIPoP Implementation





Measuring LOLLIPOP's Efficiency and Accuracy

SPEC2000 Benchmark Data

Benchmark	LOC	Matrix Size N	PPA Analysis Time [Unsafe]	PPA Analysis Time [Safe]
Bzip2	4686	251	**0.3 seconds**	**0.3 seconds**
Mcf	2429	354	**0.39 seconds**	**0.61 seconds**
Gzip	8616	563	**0.71 seconds**	**0.77 seconds**
Crafty	21297	1917	**5.49 seconds**	**5.51 seconds**
Vpr	17750	1976	**9.33 seconds**	**10.34 seconds**
Twolf	20469	2611	**16.59 seconds**	**20.64 seconds**
Parser	11402	2732	**30.72 seconds**	**50.04 seconds**
Vortex	67225	11018	**3min 59seconds**	**4min 56seconds**
Gap	71766	25882	**54min 56seconds**	**83min 38seconds**
Perlbmk	85221	20922	**44min 15seconds**	**89min 43seconds**
Gcc	22225	42109	**5hour 10 min**	**Still Running…**

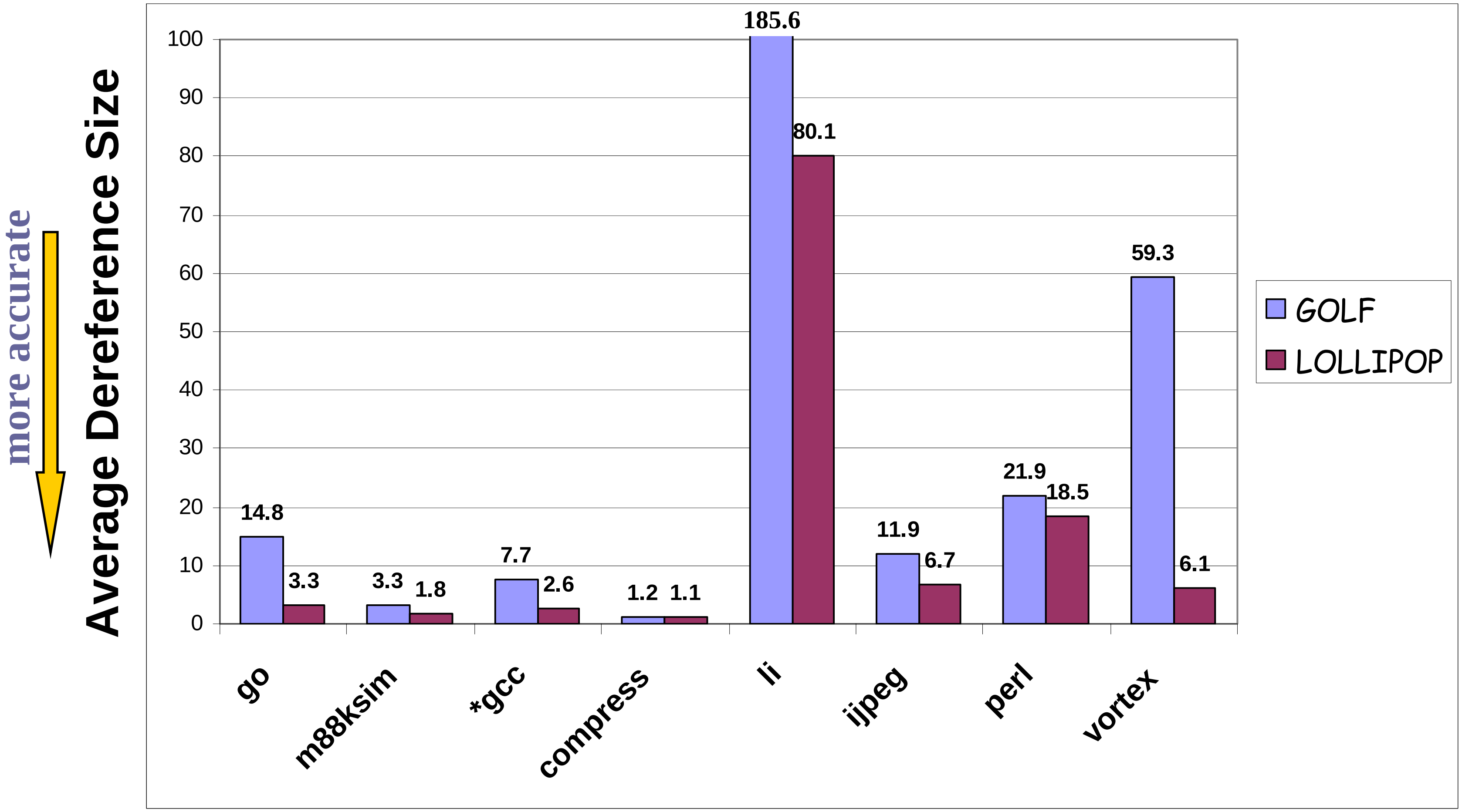
Experimental Framework: 3GHz P4 with 2GB of RAM

☞ **Scales to all of SPECint**

Comparison with Das's GOLF

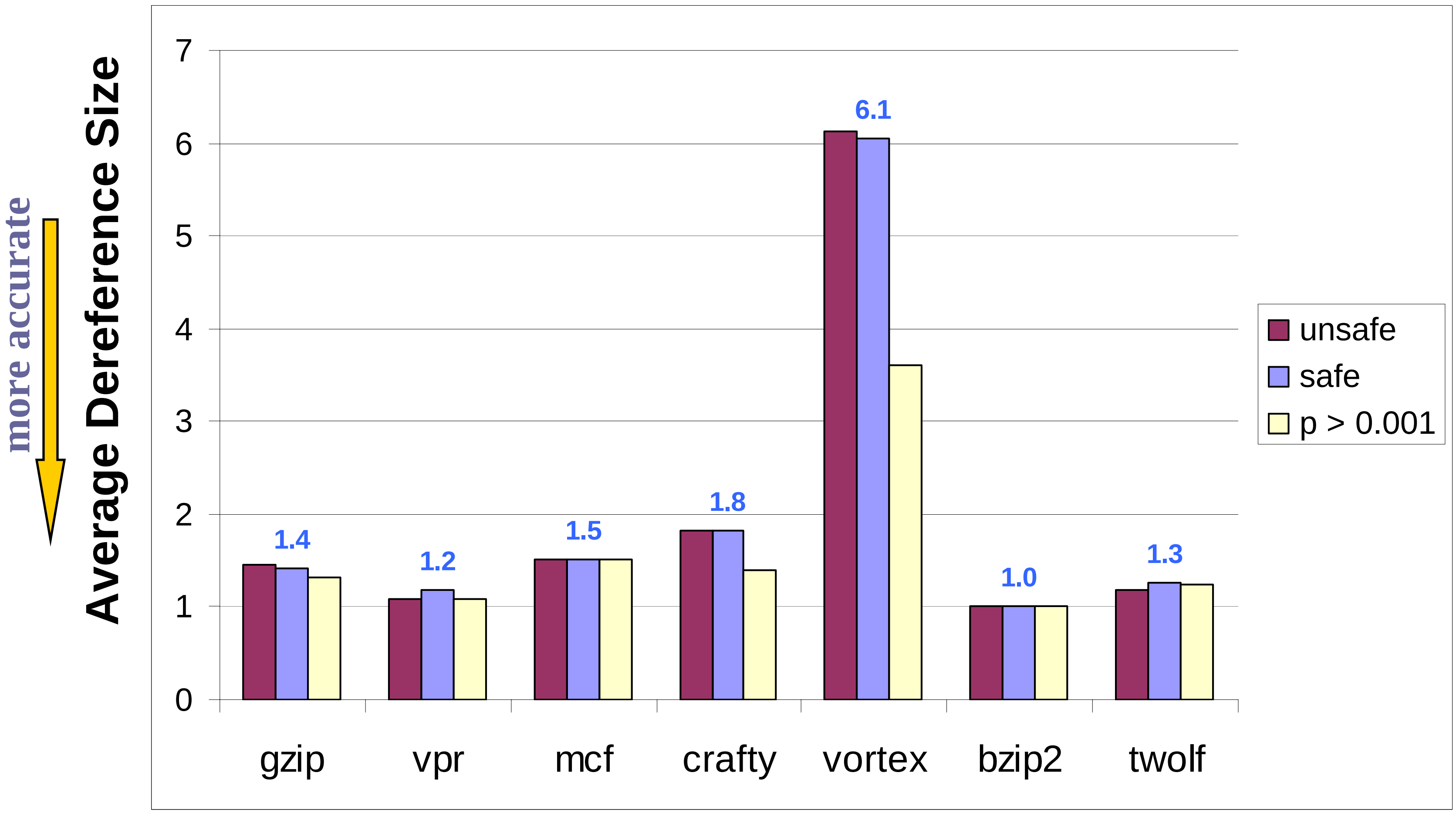
	GOLF	LOLLIPoP
Probabilistic	No	Yes
Context Sensitive	One-level	One-level
Flow Sensitive	No	Yes
Field Sensitive	No	Turned Off
Indirect Calls	Solved	Profiled
Library Calls	Modeled All	Modeled Some
Heap Model	Callsite Alloc	Callsite Alloc
Safe	Yes	Yes
Analysis Time on GCC	< 10 seconds	> 5 hours

Comparison with Das's GOLF



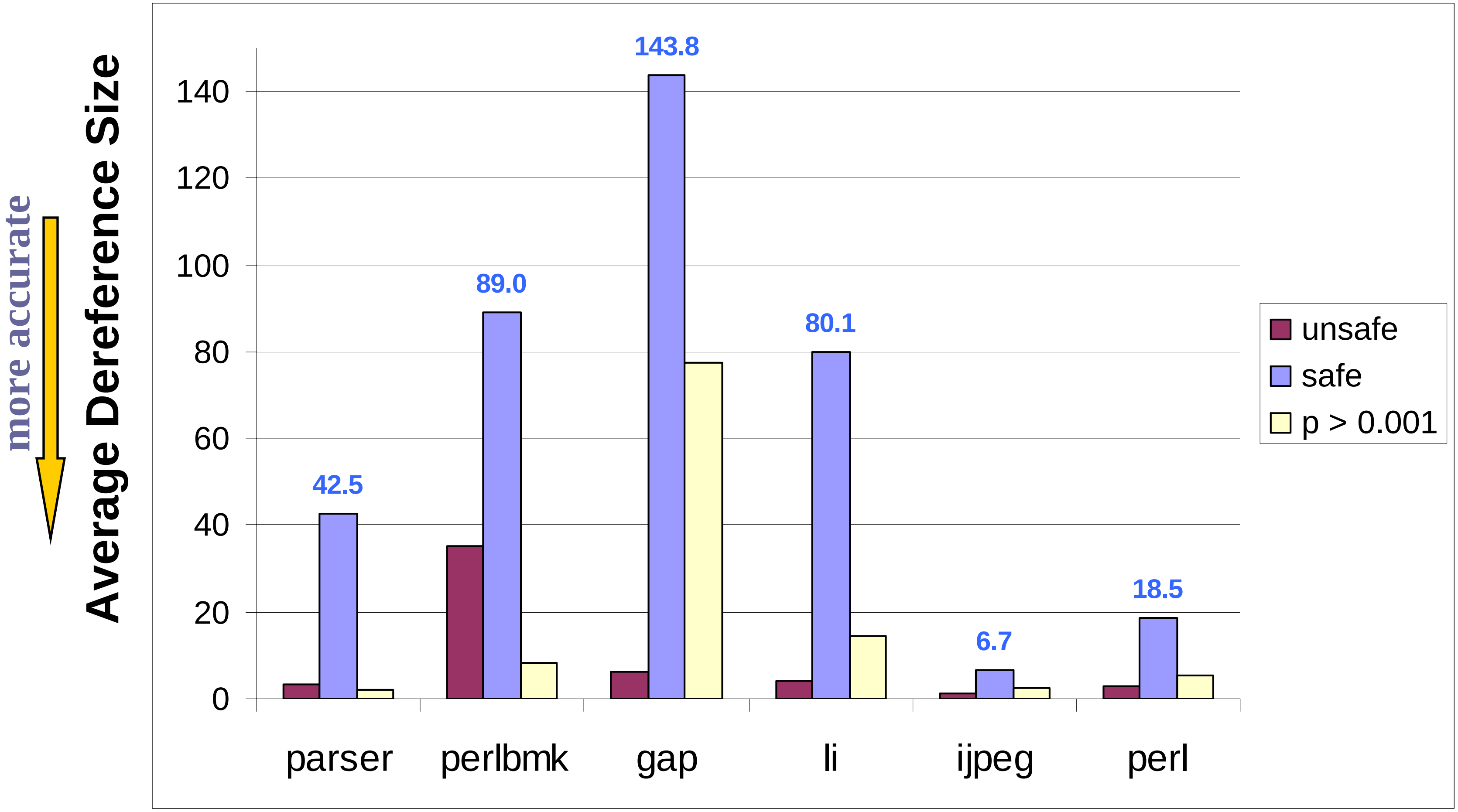
☞ **LOLLIPOP is very Accurate (even without probability information)**

Easy SPEC2000 Benchmarks



☞ **A one-level Analysis is often adequate (i.e. safe=unsafe)**

Challenging SPEC 95/2000 Benchmarks



☞ **Many improbable points-to relations can be pruned away**

Metric: Average Certainty

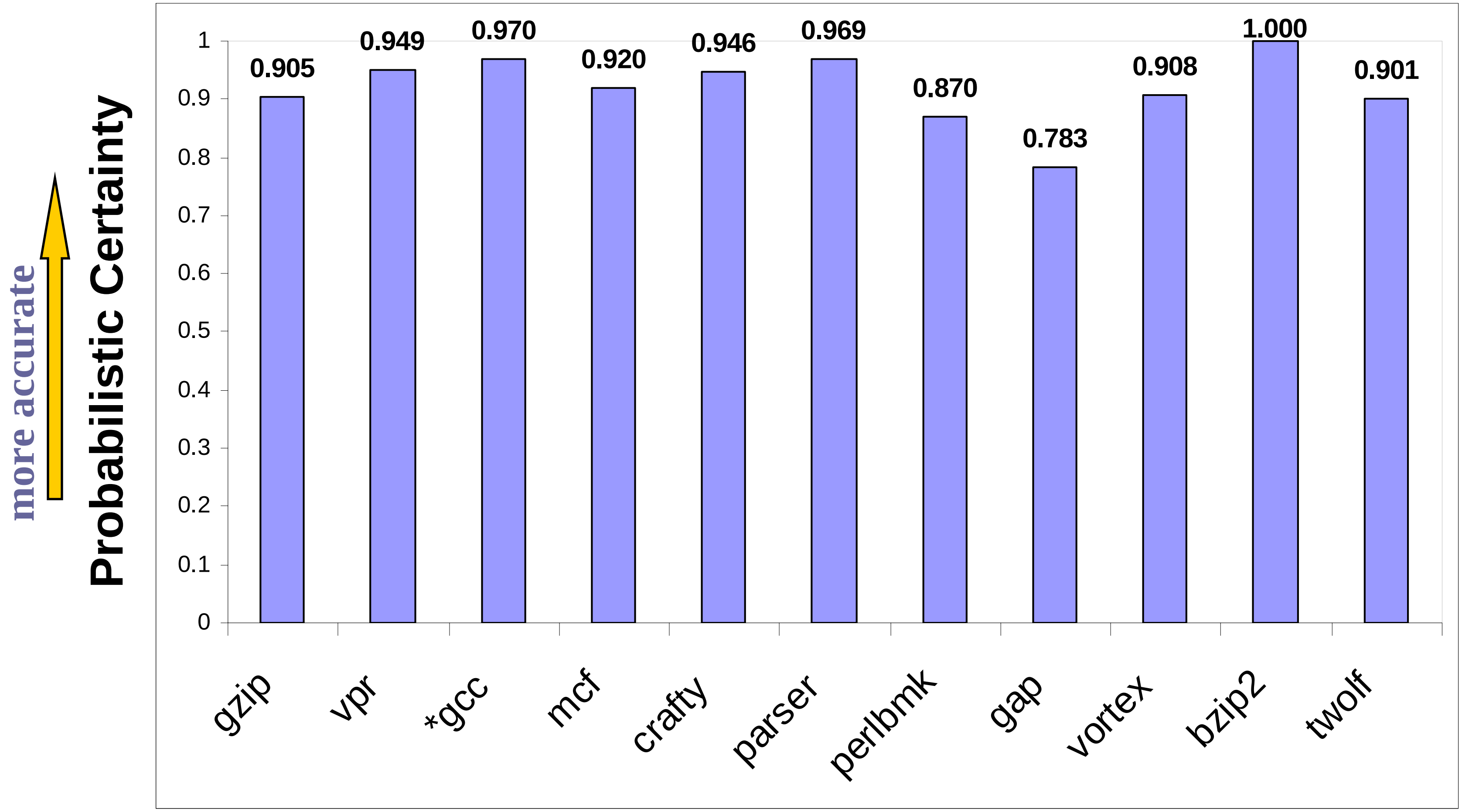
```
while(…)
{
   x = *a;
   …
}
```

Max probability value = 0.72

{ (0.72, x), (0.08, y), (0.2, z) }

$$\text{Avg Certainty} = \frac{\sum(\text{max probability value})}{(\text{num of loads \& stores})}$$

SPEC2000 Average Certainty

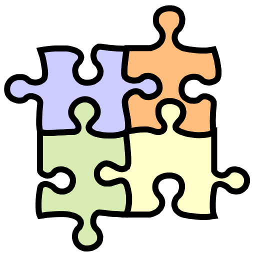


☞ **On average, *LOLLIPOP* can predict a <u>single</u> likely points-to relation**

Conclusions and Future Work

- *LOLLIPOP* 
  - A novel PPA algorithm
  - Scales to SPECint 95/2000
  - As accurate as the most precise algorithms
- **~~Future~~ Ongoing Work**
  - Measure the probabilistic accuracy
  - Optimize *LOLLIPOP*'s implementation
  - Apply PPA

☞ **Provides the key <u>puzzle piece</u> for a speculation compiler**



References

- Manuvir Das, Ben Liblit, Manuel Fahndrich, and Jakob Rehof. **Estimating the Impact of Scalable Pointer Analysis on Optimization**. SAS 2001, 260-278.
- Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. **Compiler support for speculative multithreading architecture with probabilistic points-to analysis**. PPOPP 2003, 25-36.
- Jin Lin, Tong Chen, Wei-Chung Hsu, Peng-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai and Sun Chan, **A Compiler Framework for Speculative Analysis and Optimizations**. PLDI 2003, 289-299.
- R.D. Ju, J. Collard, and K. Oukbir. **Probabilistic Memory Disambiguation and its Application to Data Speculation**. SIGARCH Comput. Archit. News 27 1999, 27-30.
- Manel Fernandez and Roger Espasa. **Speculative Alias Analysis for Executable Code**. PACT 2002, 221-231.