CSC D70: Compiler Optimization Pointer Analysis

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

The content of this lecture is adapted from the lectures of Todd Mowry, Greg Steffan, and Phillip Gibbons

Outline

- Basics
- Design Options
- Pointer Analysis Algorithms
- Pointer Analysis Using BDDs
- Probabilistic Pointer Analysis

Pros and Cons of Pointers

- Many procedural languages have pointers
  - e.g., C or C++: int *p = &x;
- Pointers are powerful and convenient
  - can build arbitrary data structures
- Pointers can also hinder compiler optimization
  - hard to know where pointers are pointing
  - must be conservative in their presence
- Has inspired much research
  - analyses to decide where pointers are pointing
  - many options and trade-offs
  - open problem: a scalable accurate analysis

Pointer Analysis Basics: Aliases

- Two variables are aliases if:
  - they reference the same memory location
- More useful:
  - prove variables reference different location

```
int x,y;
int *p = &x;
int *q = &y;
int *r = p;
int **s = &q;
```

Alias Sets ?

{x, *p, *r}

{y, *q, **s}

{q, *s}

p and q point to different locs

The Pointer Alias Analysis Problem

- Decide for every pair of pointers at every program point:
  - do they point to the same memory location?
- A difficult problem
  - shown to be undecidable by Landi, 1992
- Correctness:
  - report all pairs of pointers which do/may alias
- Ambiguous:
  - two pointers which may or may not alias
- Accuracy/Precision:
  - how few pairs of pointers are reported while remaining correct
  - i.e., reduce ambiguity to improve accuracy

Many Uses of Pointer Analysis

- Basic compiler optimizations
  - register allocation, CSE, dead code elimination, live variables, instruction scheduling, loop invariant code motion, redundant load/store elimination
- Parallelization
  - instruction-level parallelism
  - thread-level parallelism
- Behavioral synthesis
  - automatically converting C-code into gates
- Error detection and program understanding
  - memory leaks, wild pointers, security holes

Challenges for Pointer Analysis

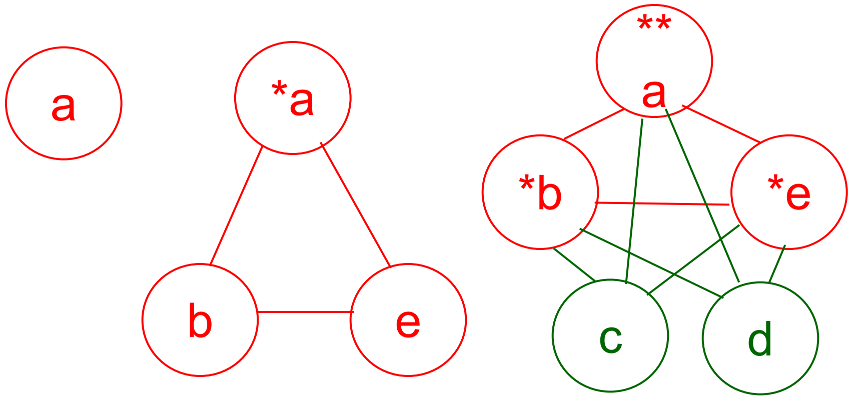
- Complexity: huge in space and time
  - compare every pointer with every other pointer
  - at every program point
  - potentially considering all program paths to that point
- Scalability vs. accuracy trade-off
  - different analyses motivated for different purposes
  - many useful algorithms (adds to confusion)
- Coding corner cases
  - pointer arithmetic (*p++), casting, function pointers, long-jumps
- Whole program?
  - most algorithms require the entire program
  - library code? optimizing at link-time only?

Pointer Analysis: Design Options

- Representation
- Heap modeling
- Aggregate modeling
- Flow sensitivity
- Context sensitivity

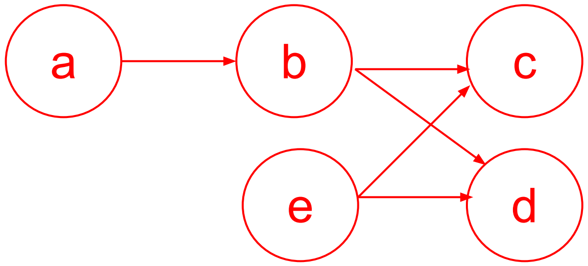
Alias Representation

- Track **pointer** aliases
  - <*a, b>, <*a, e>, <b, e>
    <**a, c>, <**a, d>, …
  - More precise, less efficient



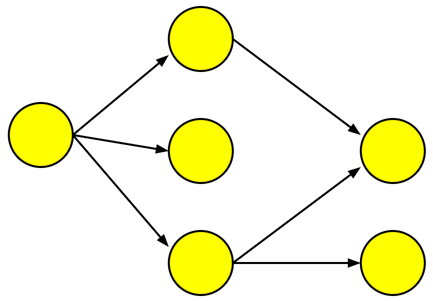
- Track **points-to** info
  - <a, b>, <b, c>, <b, d>,
    <e, c>, <e, d>
  - Less precise, more efficient
  - Why?

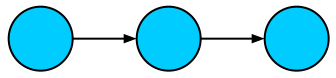
```
a = &b;
b = &c;
b  = &d;
e = b;
```



Heap Modeling Options

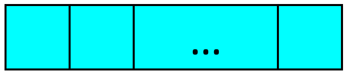
- Heap merged
  - i.e. "no heap modeling"
- Allocation site (any call to malloc/calloc)
  - Consider each to be a unique location
  - Doesn't differentiate between multiple objects allocated by the same allocation site
- Shape analysis
  - Recognize linked lists, trees, DAGs, etc.





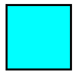
Aggregate Modeling Options

Arrays



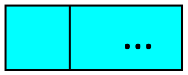
Elements are treated as individual locations

or



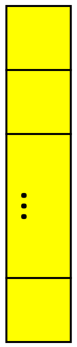
Treat entire array as a single location

or



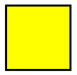
Treat first element separate from others

Structures



Elements are treated as individual locations ("field sensitive")

or



Treat entire structure as a single location

What are the tradeoffs?

Flow Sensitivity Options

- Flow insensitive
  - The order of statements doesn't matter
    - Result of analysis is the same regardless of statement order
  - Uses a single global state to store results as they are computed
  - Not very accurate
- Flow sensitive
  - The order of the statements matter
  - Need a control flow graph
  - Must store results for each program point
  - Improves accuracy
- Path sensitive
  - Each path in a control flow graph is considered

Flow Sensitivity Example

(assuming allocation-site heap modeling)

```
S1: a = malloc(…);
S2: b = malloc(…);
S3: a = b;
S4: a = malloc(…);
S5: if(c)
       a = b;
S6: if(!c)
       a = malloc(…);
S7: … = *a;
```

Flow Insensitive

a_{S7} ⍰ {heapS1, heapS2, heapS4, heapS6}

(order doesn't matter, union of all possibilities)

Flow Sensitive

a_{S7} ⍰ {heapS2, heapS4, heapS6}

(in-order, doesn't know s5 & s6 are exclusive)

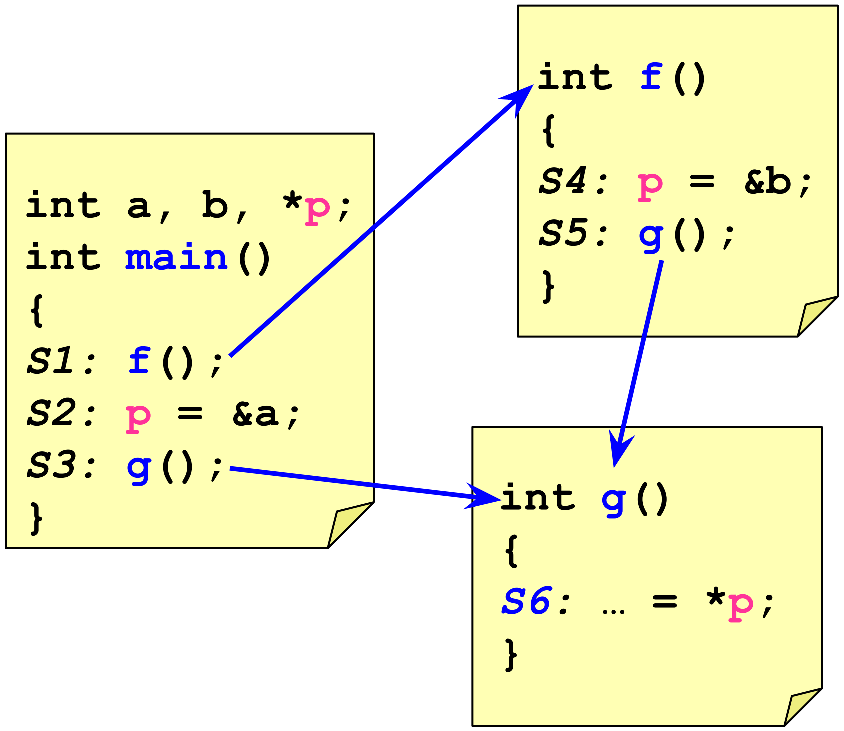
Path Sensitive

a_{S7} ⍰ {heapS2, heapS6}

(in-order, knows s5 & s6 are exclusive)

Context Sensitivity Options

- Context insensitive/sensitive
  - whether to consider different calling contexts
  - e.g., what are the possibilities for `p` at `S6`?



```
int a, b, *p;
int main()
{
S1: f();
S2: p = &a;
S3: g();
}
```

```
int f()
{
S4: p = &b;
S5: g();
}
```

```
int g()
{
S6: … = *p;
}
```

Context Insensitive:

p_{S6} => {a,b}

Context Sensitive:

Called from S5:p_{S6} => {b}
Called from S3:p_{S6} => {a}

Pointer Alias Analysis Algorithms

References:

- *"Points-to analysis in almost linear time"*, Steensgaard, POPL 1996
- *"Program Analysis and Specialization for the C Programming Language"*, Andersen, Technical Report, 1994
- *"Context-sensitive interprocedural points-to analysis in the presence of function pointers"*, Emami et al., PLDI 1994
- *"Pointer analysis: haven't we solved this problem yet?"*, Hind, PASTE 2001
- *"Which pointer analysis should I use?"*, Hind et al., ISSTA 2000
- …

- "*Introspective analysis: context-sensitivity, across the board*", Smaragdakiset al., PLDI 2014
- "*Sparse flow-sensitive pointer analysis for multithreaded programs*", Sui et al., CGO 2016
- "*Symbolic range analysis of pointers*", Paisanteet al., CGO 2016

Address Taken

- Basic, fast, ultra-conservative algorithm
  - flow-insensitive, context-insensitive
  - often used in production compilers
- Algorithm:
  - Generate the set of all variables whose addresses are assigned to another variable.
  - Assume that any pointer can potentially point to any variable in that set.
- Complexity: O(n) - linear in size of program
- Accuracy: very imprecise

Address Taken Example

```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: … = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
     T local;
     if(…)
s9:      p = &local;
}
```

p_{S5} = {heap_S1, p, heap_S4, heap_S6, q, heap_S8, local}

Andersen's Algorithm

- Flow-insensitive, context-insensitive, iterative
- Representation:
  - one points-to graph for entire program
  - each node represents exactly one location
- For each statement, build the points-to graph:

`y = &x`	`y` points-to `x`
`y = x`	if `x` points-to `w` then `y` points-to `w`
`*y = x`	if `y` points-to `z` and `x` points-to `w` then `z` points-to `w`
`y = *x`	if `x` points-to `z` and `z` points-to `w` then `y` points-to `w`

- Iterate until graph no longer changes
- Worst case complexity: $O(n^3)$, where n = program size

Andersen Example

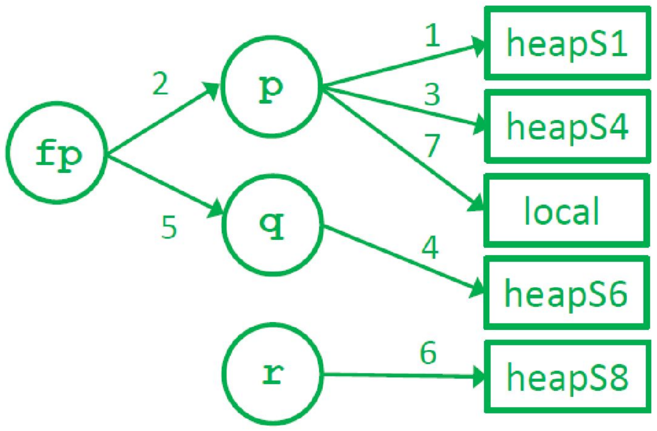
```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: … = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
    T local;
    if(…)
s9:     p = &local;
}
```

p_{S5} = {heap_S1, heap_S4, local}



Steensgaard’s Algorithm

- Flow-insensitive, context-insensitive
- Representation:
  - a compact points-to graph for entire program
    - each node can represent multiple locations
    - but can only point to one other node
      - i.e. every node has a fan-out of 1 or 0
- *union-find* data structure implements fan-out
  - “unioning” while finding eliminates need to iterate
- Worst case complexity: O(n)
- Precision: less precise than Andersen’s

Steensgaard Example

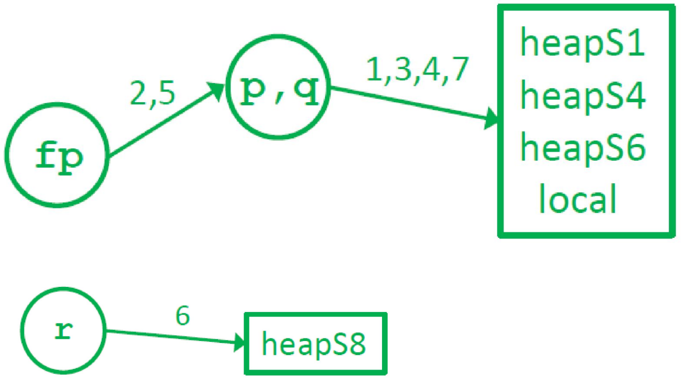
```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: … = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
    T local;
    if(…)
s9:     p = &local;
}
```

p_{S5} = {heap_S1, heap_S4, heap_S6, local}



Example with Flow Sensitivity

```
T *p, *q, *r;

int main() {
S1: p = alloc(T);
    f();
    g(&p);
S4: p = alloc(T);
S5: … = *p;
}
```

```
void f() {
S6: q = alloc(T);
    g(&q);
S8: r = alloc(T);
}
```

```
g(T **fp) {
    T local;
    if(…)
s9:     p = &local;
}
```

p_{S5} = {heap_S4}

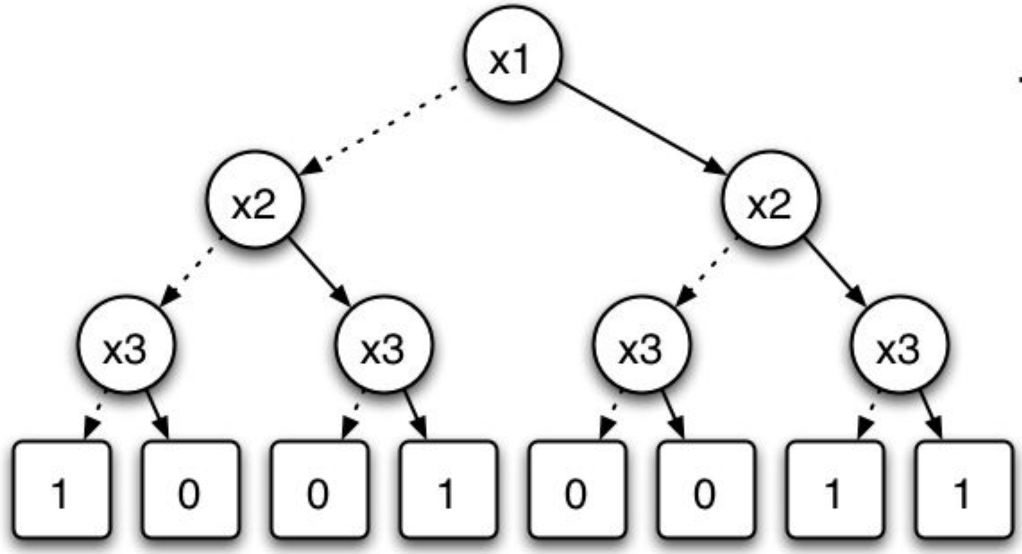
p_{S9} = {local, heap_s1}

Pointer Analysis Using BDDs: Binary Decision Diagrams

References:

- *"Cloning-based context-sensitive pointer alias analysis using binary decision diagrams"*, Whaley and Lam, PLDI 2004
- *"Symbolic pointer analysis revisited"*, Zhu and Calman, PDLI 2004
- *"Points-to analysis using BDDs"*, Berndl et al, PDLI 2003

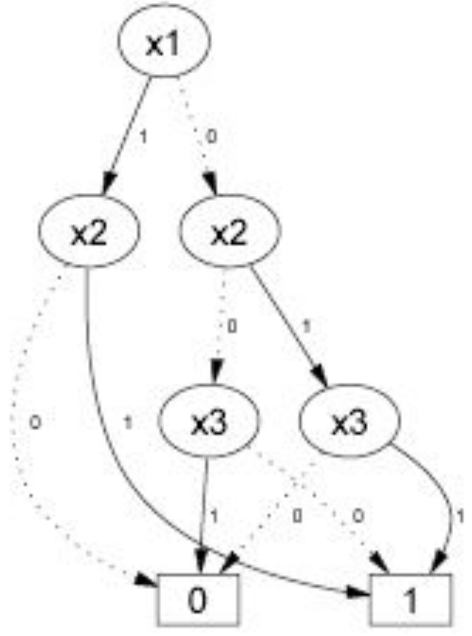
Binary Decision Diagram (BDD)



Binary Decision Tree

x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Truth Table



BDD

BDD-Based Pointer Analysis

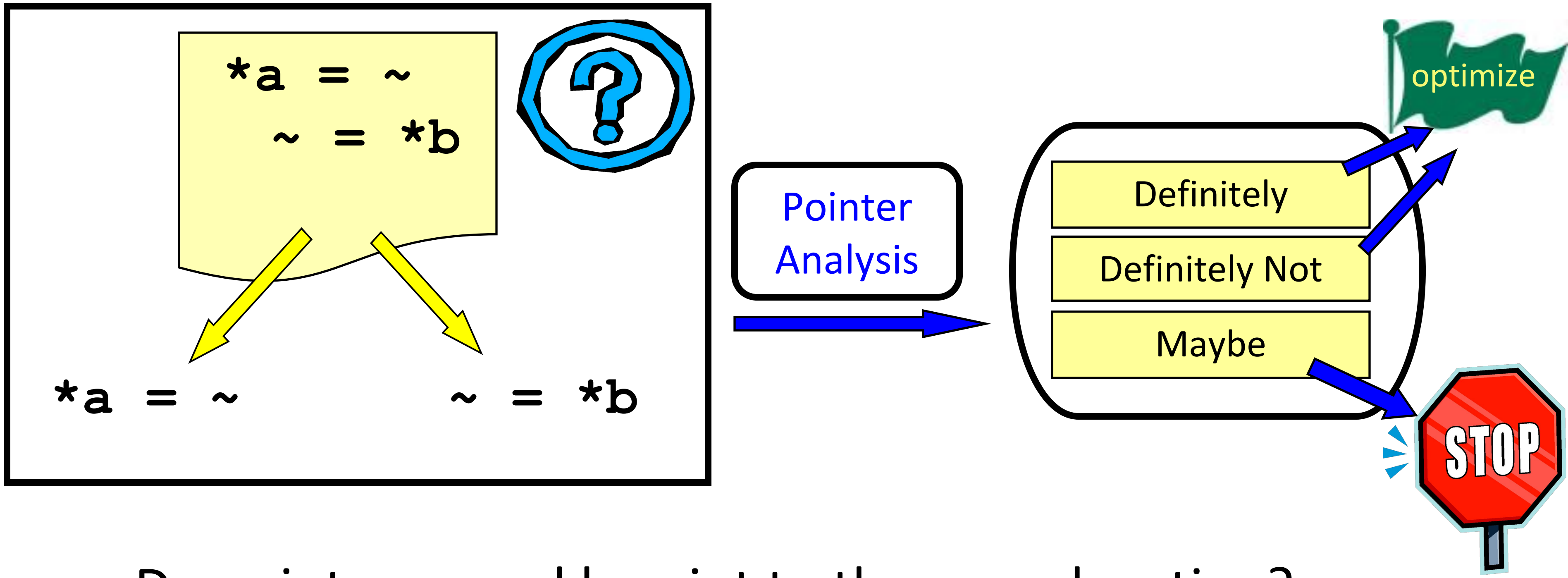
- Use a BDD to represent transfer functions
  - encode procedure as a function of its calling context
  - compact and efficient representation
- Perform context-sensitive, inter-procedural analysis
  - similar to dataflow analysis
  - but across the procedure call graph
- Gives accurate results
  - and scales up to large programs

Probabilistic Pointer Analysis

<u>References</u>:

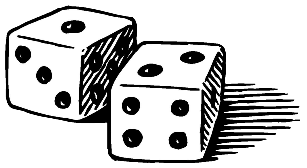
- *"A Probabilistic Pointer Analysis for Speculative Optimizations"*, DaSilva and Steffan, ASPLOS 2006
- *"Compiler support for speculative multithreading architecture with probabilistic points-to analysis"*, Shen et al., PPoPP 2003
- *"Speculative Alias Analysis for Executable Code"*, Fernandez and Espasa, PACT 2002
- *"A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion"*, Dai et al., CGO 2005
- *"Speculative register promotion using Advanced Load Address Table (ALAT)"*, Lin et al., CGO 2003

Pointer Analysis: Yes, No, & Maybe



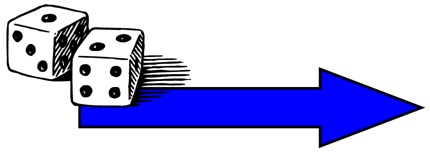
- Do pointers a and b point to the same location?
  - Repeat for every pair of pointers at every program point
- How can we optimize the "maybe" cases?

Let’s Speculate



- Implement a potentially unsafe optimization
  - Verify and Recover if necessary

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



a is *probably* loop invariant

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

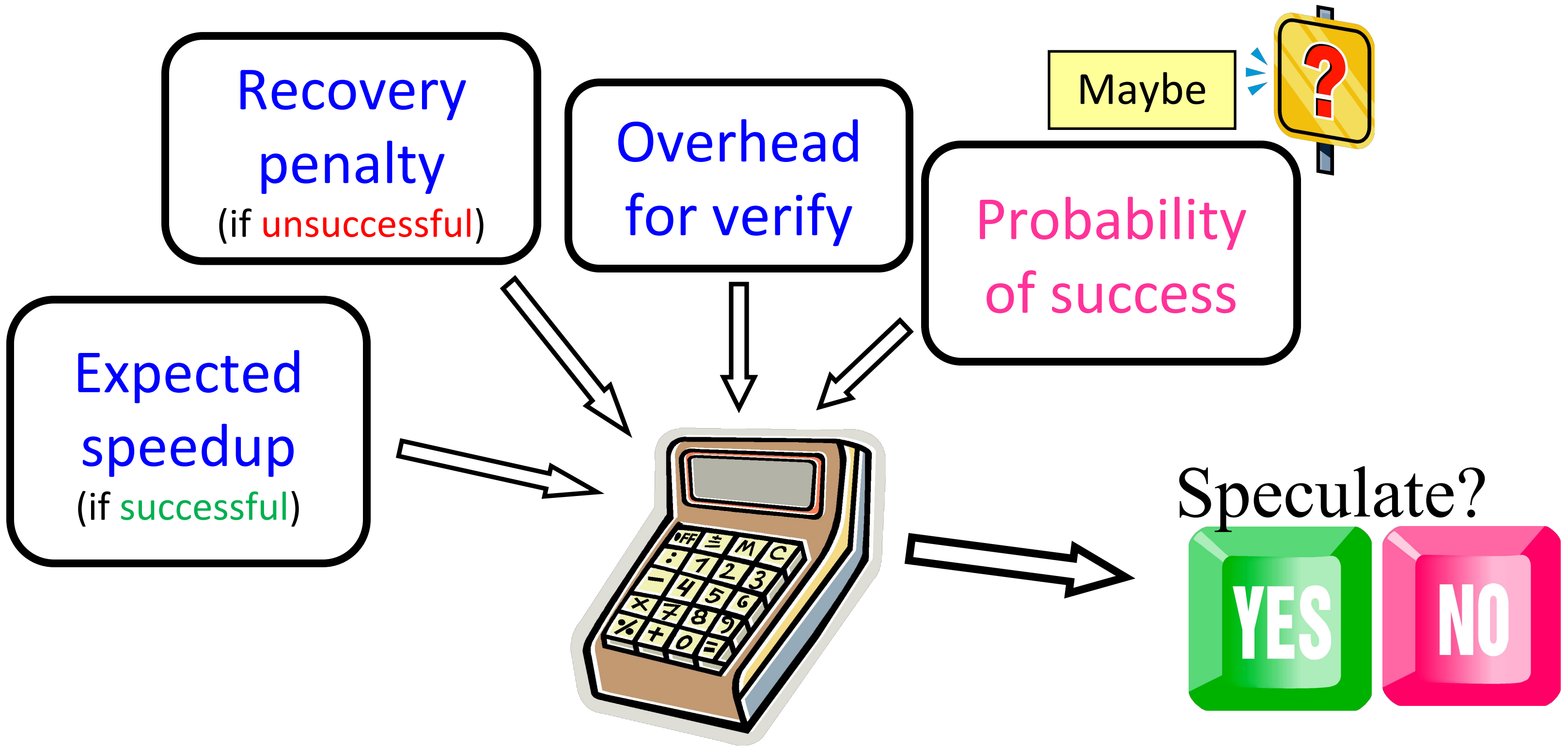
Data Speculative Optimizations

- EPIC Instruction sets
  - Support for speculative load/store instructions (e.g., Itanium)
- Speculative compiler optimizations
  - Dead store elimination, redundancy elimination, copy propagation, strength reduction, register promotion
- Thread-level speculation (TLS)
  - Hardware and compiler support for speculative parallel threads
- Transactional programming
  - Hardware and software support for speculative parallel transactions

Heavy reliance on detailed profile feedback

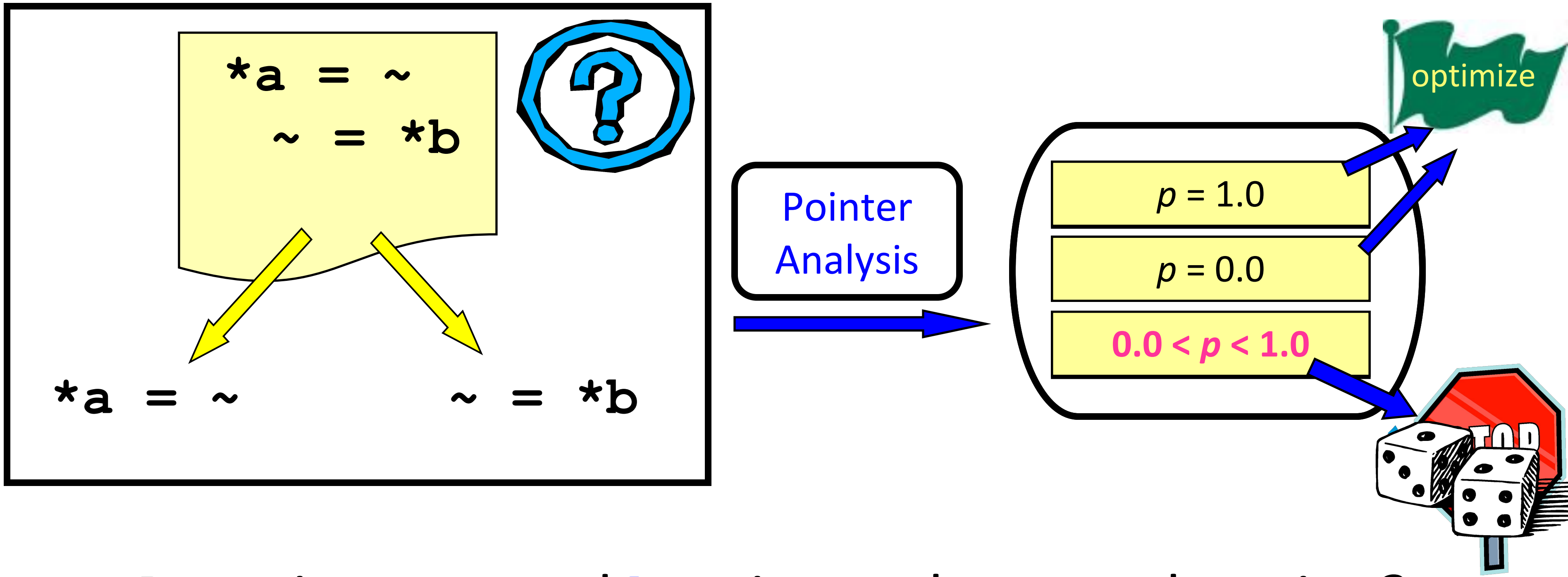
Can We Quantify "Maybe"?

- Estimate the potential benefit for speculating:


Recovery penalty (if unsuccessful)
Overhead for verify
Maybe
Probability of success
Expected speedup (if successful)
Speculate?
YES
NO

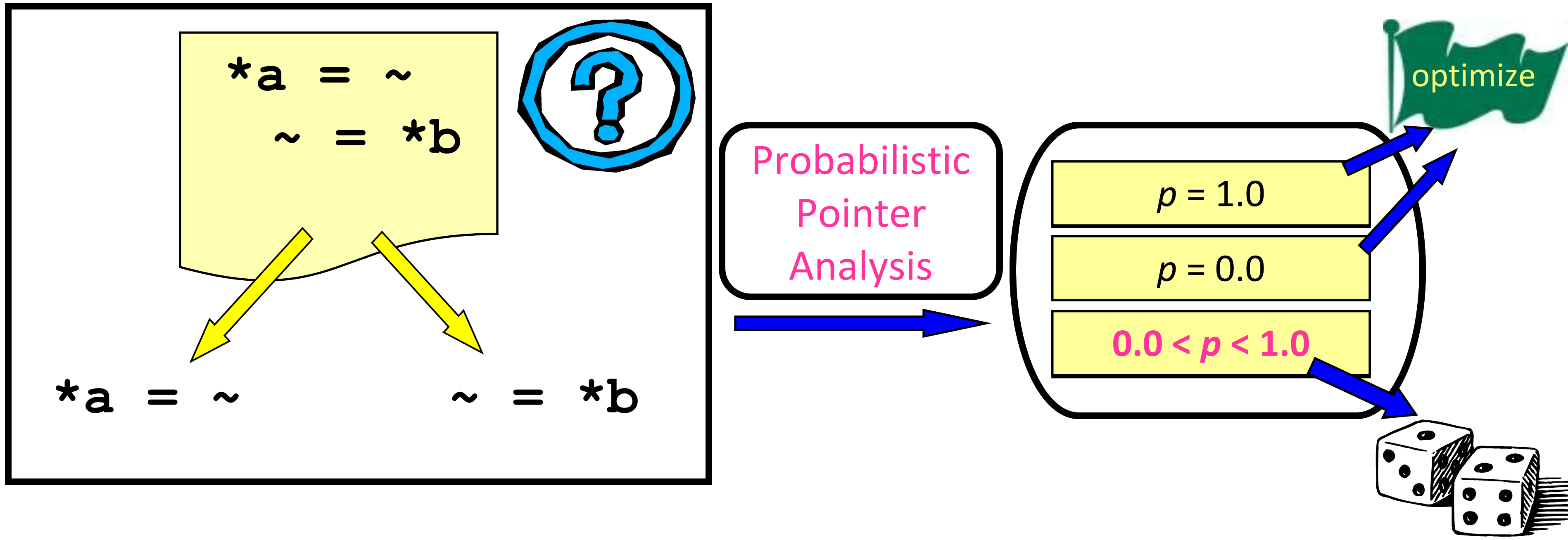
Ideally "maybe" should be a probability.

Conventional Pointer Analysis



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

Probabilistic Pointer Analysis



- Potential advantage of Probabilistic Pointer Analysis:
  - it doesn’t need to be safe

PPA Research Objectives

- Accurate points-to probability information
  - at every static pointer dereference
- Scalable analysis
  - Goal: 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)

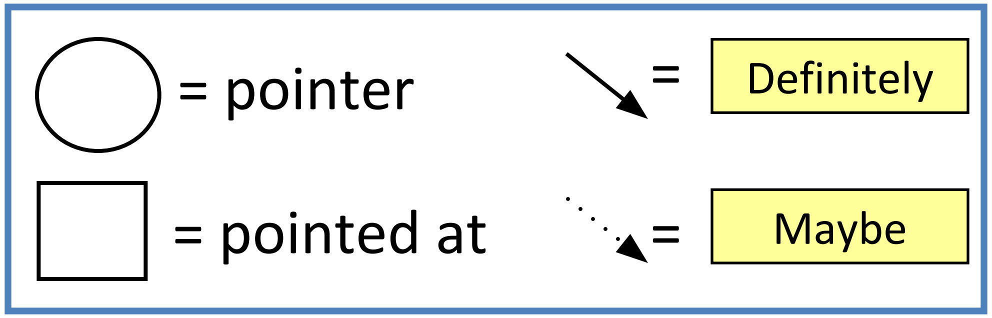
Traditional Points-To Graph

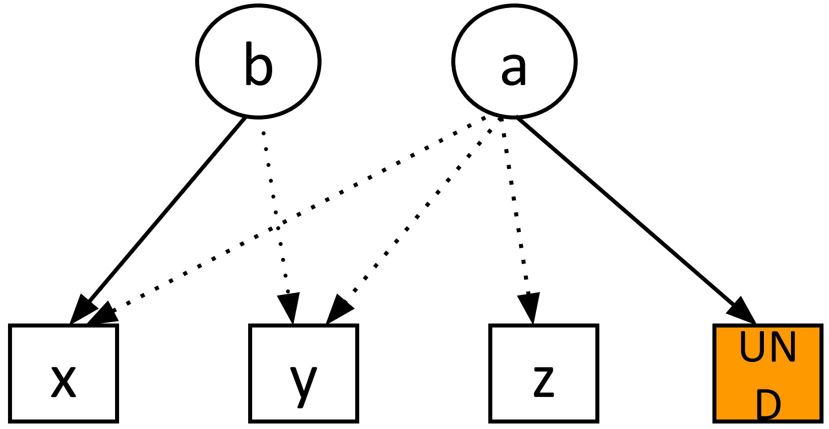
```
int x, y, z, *b = &x;
void foo(int *a) {

   if(…)
      b = &y;

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

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





Results are inconclusive

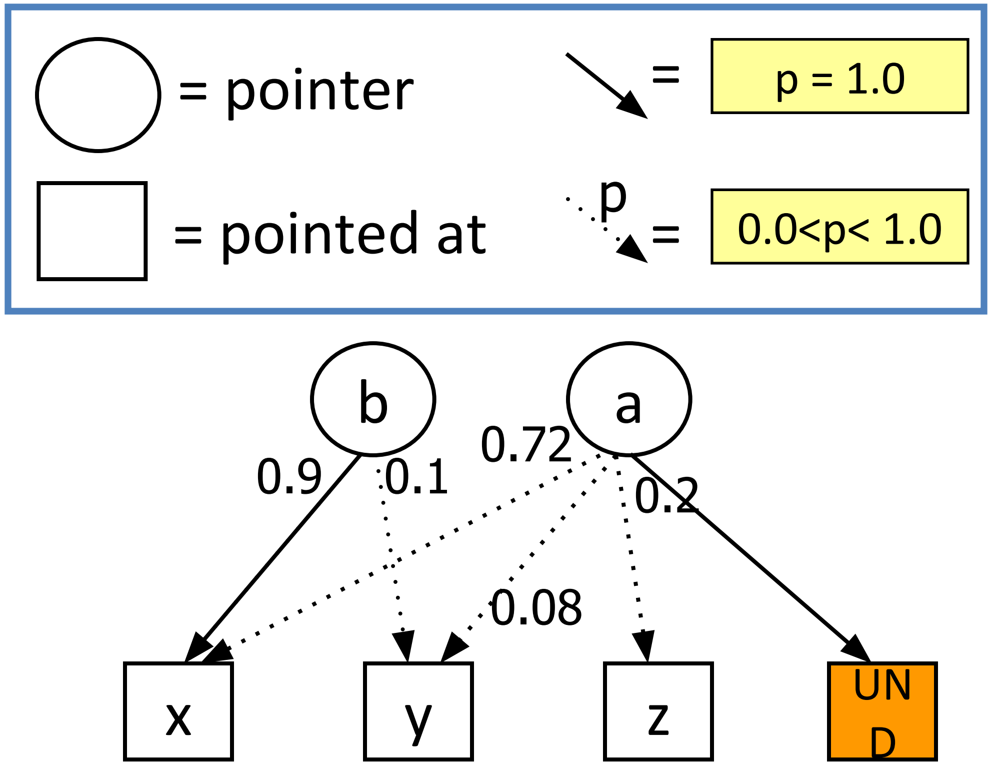
Probabilistic Points-To Graph

```
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

Probabilistic Pointer Analysis Results Summary

- Matrix-based, transfer function approach
  - SUIF/Matlab implementation
- Scales to the SPECint 95/2000 benchmarks
  - One-level context and flow sensitive
- As accurate as the most precise algorithms
- Interesting result:
  - ~90% of pointers tend to point to only one thing

Pointer Analysis Summary

- Pointers are hard to understand at compile time!
  - accurate analyses are large and complex
- Many different options:
  - Representation, heap modeling, aggregate modeling, flow sensitivity, context sensitivity
- Many algorithms:
  - Address-taken, Steensgarde, Andersen, Emami
  - BDD-based, probabilistic
- Many trade-offs:
  - space, time, accuracy, safety
- Choose the right type of analysis given how the information will be used

CSC D70: Compiler Optimization Memory Optimizations (Intro)

Prof. Gennady Pekhimenko

University of Toronto

Winter 2019

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons

Caches: A Quick Review

- How do they work?
- Why do we care about them?
- What are typical configurations today?
- What are some important cache parameters that will affect performance?

Optimizing Cache Performance

- Things to enhance:
  - temporal locality
  - spatial locality

- Things to minimize:
  - conflicts (i.e. bad replacement decisions)

What can the *compiler* do to help?

Two Things We Can Manipulate

- Time:
  - When is an object accessed?

- Space:
  - Where does an object exist in the address space?

How do we exploit these two levers?

Time: Reordering Computation

- What makes it difficult to know *when* an object is accessed?
- How can we predict a better time to access it?
  - What information is needed?
- How do we know that this would be safe?

Space: Changing Data Layout

- What do we know about an object's location?
  - scalars, structures, pointer-based data structures, arrays, code, etc.
- How can we tell what a better layout would be?
  - how many can we create?
- To what extent can we safely alter the layout?

Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

Scalars

- Locals
- Globals
- Procedure arguments
- Is cache performance a concern here?
- If so, what can be done?

```
int x;
double y;
foo(int a){
   int i;
   …
   x = a*i;
   …
}
```

Structures and Pointers

- What can we do here?
  - within a node
  - across nodes

```
struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
} node;
```

- What limits the compiler's ability to optimize here?

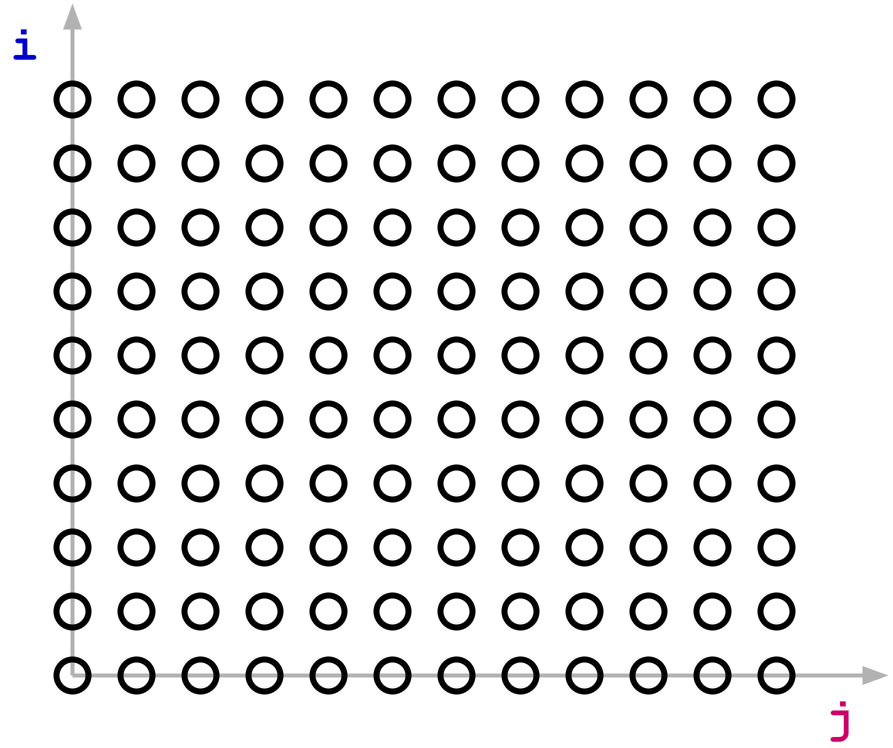
Arrays

```
double A[N][N], B[N][N];
…
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```

- usually accessed within loops nests
  - makes it easy to understand “time”
- what we know about array element addresses:
  - start of array?
  - relative position within array

Handy Representation: "Iteration Space"

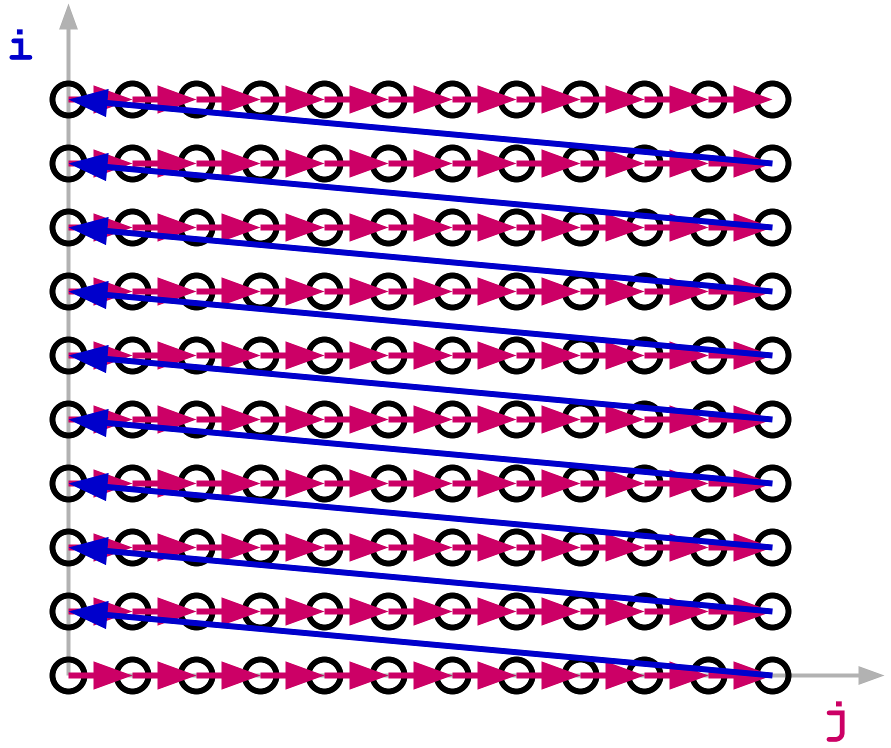
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```



- each position represents an iteration

Visitation Order in Iteration Space

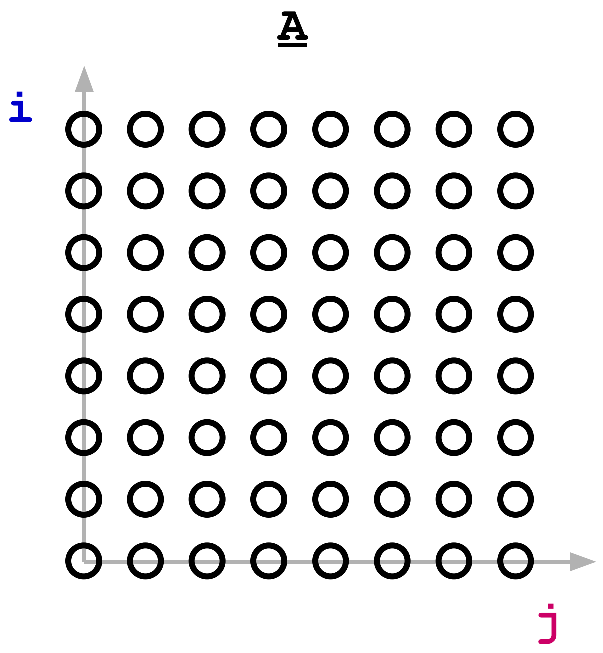
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] =
B[j][i];
```

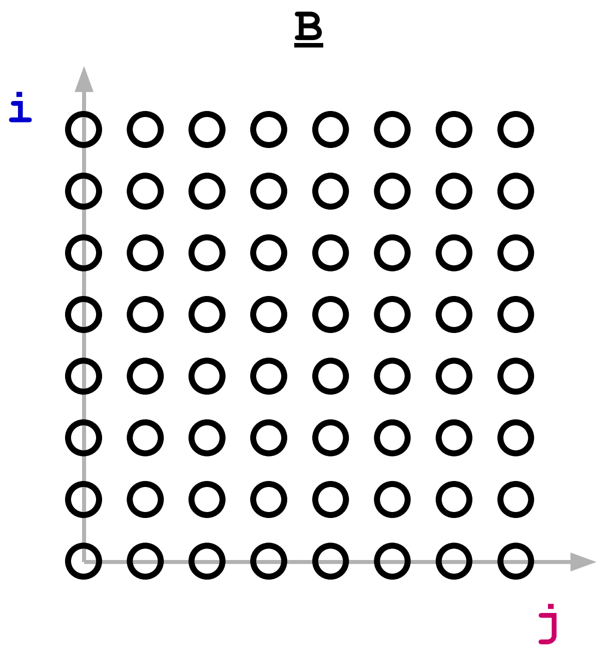


- Note: iteration space ≠ data space

When Do Cache Misses Occur?

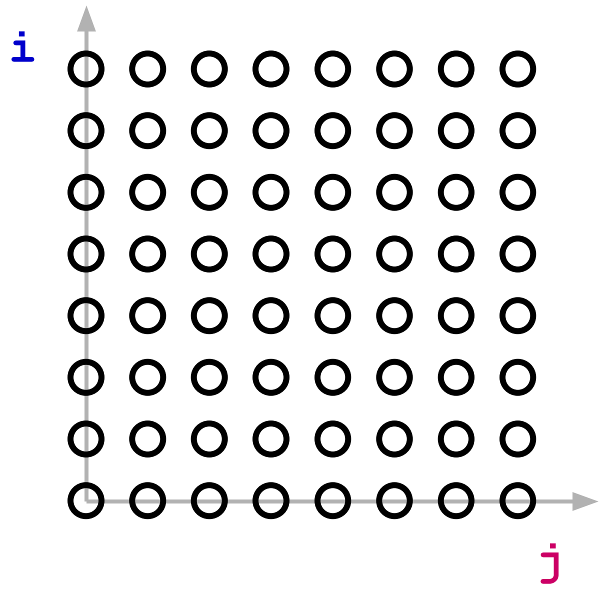
```
for i = 0 to N-1
    for j = 0 to N-1
        A[i][j] = B[j][i];
```





When Do Cache Misses Occur?

```
for i = 0 to N-1
    for j = 0 to N-1
        A[i+j][0] = i*j;
```



Optimizing the Cache Behavior of Array Accesses

- We need to answer the following questions:
  - when do cache misses occur?
    - use “locality analysis”
  - can we change the order of the iterations (or possibly data layout) to produce better behavior?
    - evaluate the cost of various alternatives
  - does the new ordering/layout still produce correct results?
    - use “dependence analysis”

Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- …

(we will briefly discuss the first two next week)

CSC D70: Compiler Optimization Pointer Analysis & Memory Optimizations (Intro)

Prof. Gennady Pekhimenko
University of Toronto

Winter 2019

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons