Code analysis *and* transformation

# Dependences

Simone Campanoni
simone.campanoni@northwestern.edu

# Dependences: the big picture

- Code transformations are designed
  to preserve the semantics of the code given as input
  - As defined earlier, semantics of a program is the Input=>Output mapping

```
1: varX = par1 + 1
2: varY = par2 + par1
3: varZ = varY + varX
4: print(varZ)
```

```
4: print(varZ)
1: varX = par1 + 1
2: varY = par2 + par1
3: varZ = varX + varY
```

```
2: varY = par2 + par1
1: varX = par1 + 1
3: varZ = varX + varY
4: print(varZ)
```

- A dependence A -> B is satisfied if
  A will always execute before B
- If we satisfy all dependences in the code,
  then we will preserve I => O

```
A: varX = 1;
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```

# Outline

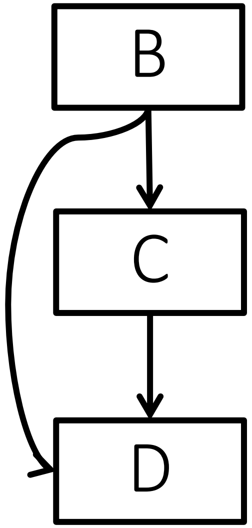
- Control dependences
- Data dependences
- Introduction to memory alias analysis

# Control dependence intuition

- Dependence: C will be executed depending on B

- How to identify C? (automatically)
  - Do we need a DFA?
  - We need a Control Flow Analysis

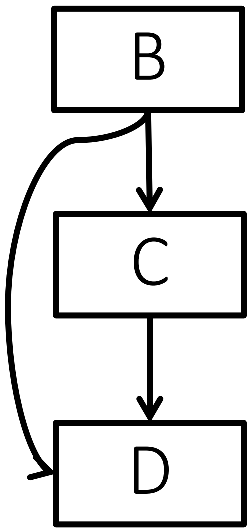
```
A: varX = 1;
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```



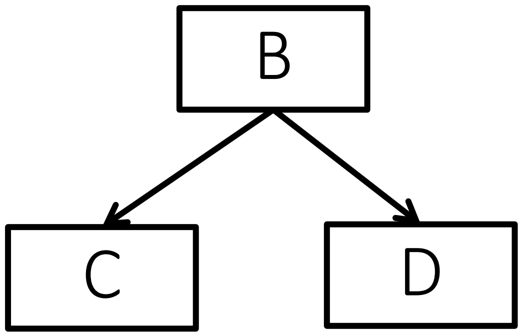

CFG

# Dominators

**Definition:** Node *d* dominates node *n* in a graph
if every path from the start node to *n* goes through *d*




CFG




Immediate dominator tree

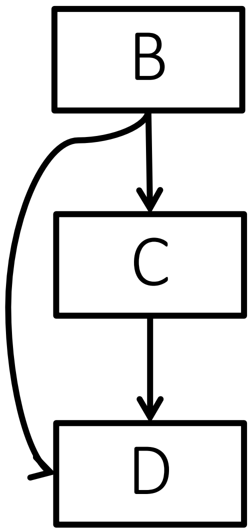
```
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```

**Are dominators useful to identify the control dependence between C and B?**

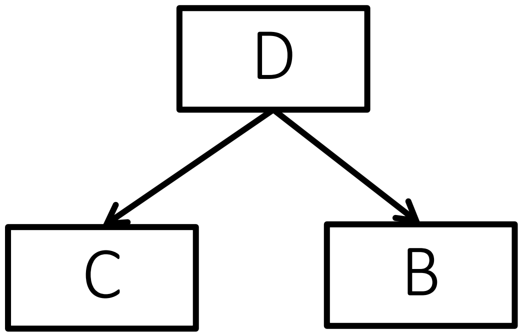
# Post-Dominators

**Assumption:** Single exit node in CFG

**Definition:** Node *d* post-dominates node *n* in a graph if every path from *n* to the exit node goes through *d*




CFG




Immediate post-dominator tree

```
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```

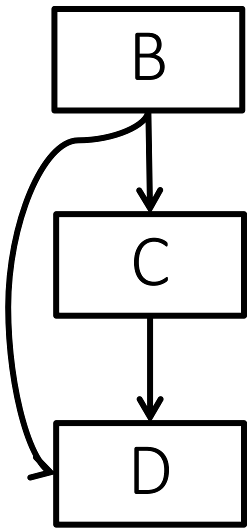
**How can we identify C and B with the post-dominator tree and the CFG?**

***B determines whether C executes or not***

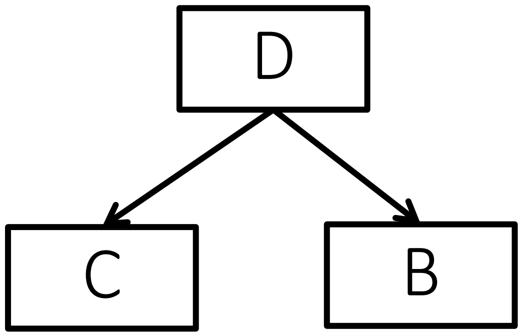
# Control dependence in our example

Node *C* is control-dependent on *B* because

1. C is the successor of B
2. C does not post-dominate B




CFG




Immediate post-dominator tree

```
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```

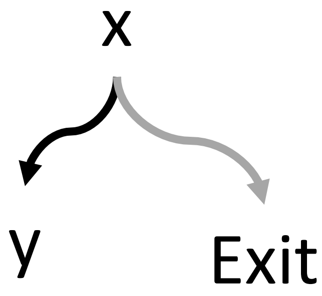
**How can we identify C and B with the post-dominator tree and the CFG?**

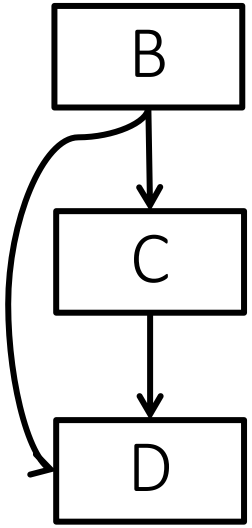
***B determines whether C executes or not***

# Control dependences (almost correct)

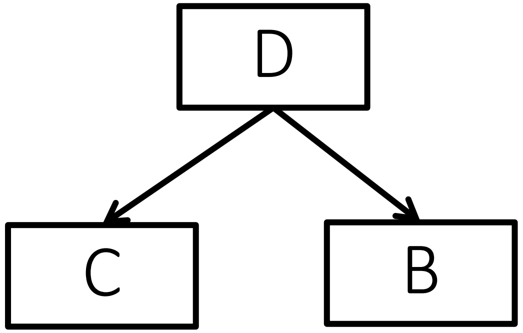
A node *Y* control-depends on another node *X* if and only if

1. There is a path from X to Y such that
   every node in that path other than X is post-dominated by Y
2. X is not post-dominated by Y




CFG




Immediate
post-dominator tree

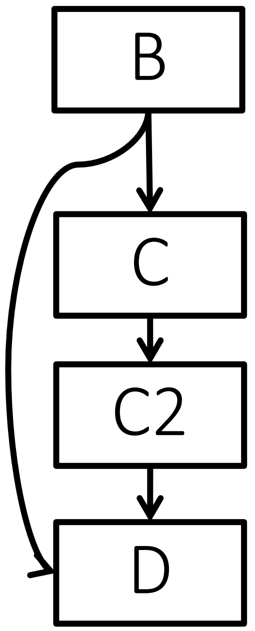
```
B: if (par1 > 5)
C:     varX = par1 + 1
D: print(varX)
```

Why?

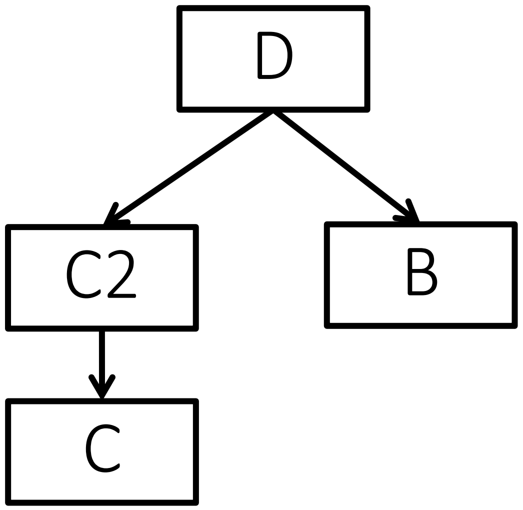
# Control dependences (almost correct)

A node *Y* control-depends on another node *X* if and only if

1. There is a path from X to Y such that
   every node in that path other than X is post-dominated by Y
2. X is not post-dominated by Y




CFG




Immediate
post-dominator tree

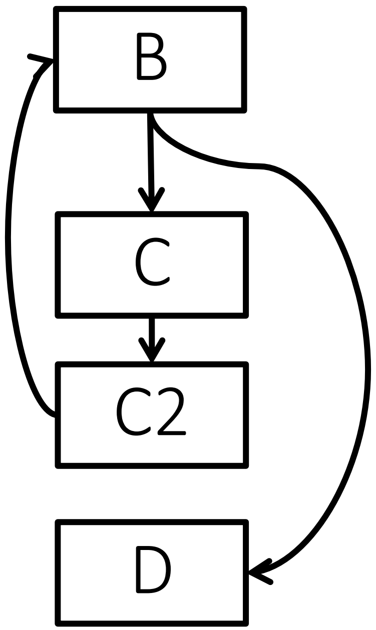
```
B: if (par1 > 5)
C:     varX = par1 + 1
C2:    ...
D: print(varX)
```

Why?

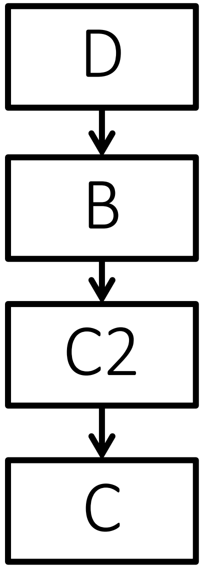
# Control dependences (almost correct)

A node *Y* control-depends on another node *X* if and only if

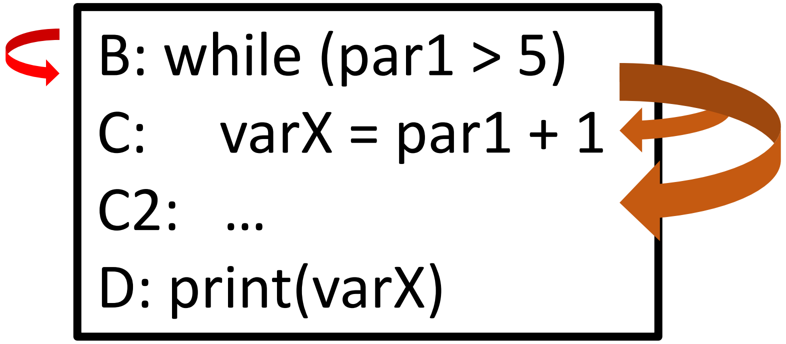
1. There is a path from X to Y such that every node in that path other than X is post-dominated by Y
2. X is not post-dominated by Y




CFG



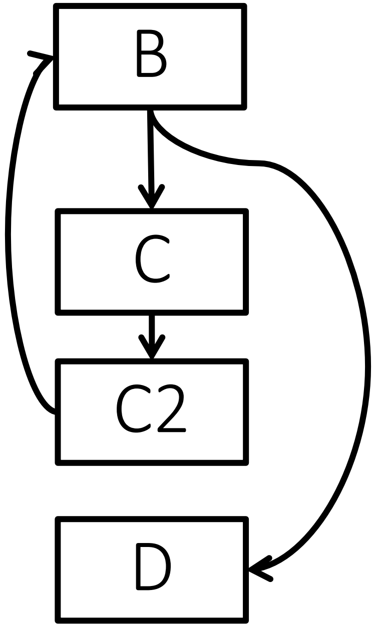

Immediate post-dominator tree



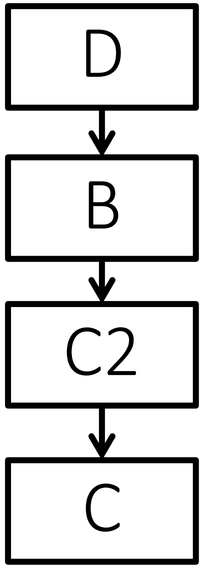
# Control dependences

A node *Y* control-depends on another node *X* if and only if

1. There is a path from X to Y such that every node in that path other than X is post-dominated by Y
2. X is not **strictly** post-dominated by Y




CFG

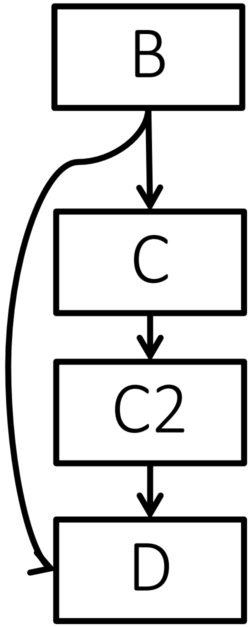



Immediate post-dominator tree

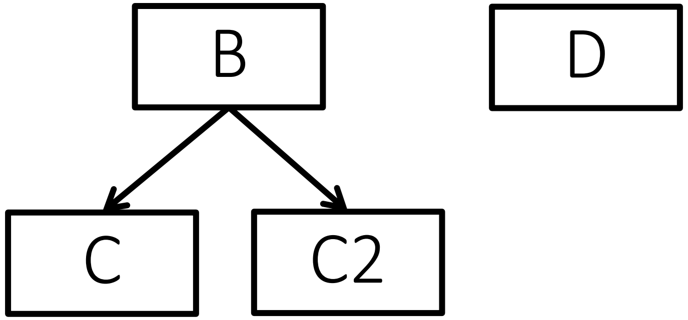
```
B: while (par1 > 5)
C:     varX = par1 + 1
C2:   ...
D: print(varX)
```

# Control dependence graph (CDG)

- Graph (N, E) where
  - N are basic blocks
  - Exist an edge (x,y) in E if and only if y control-depends on x




CFG




CDG

An use of CDG:

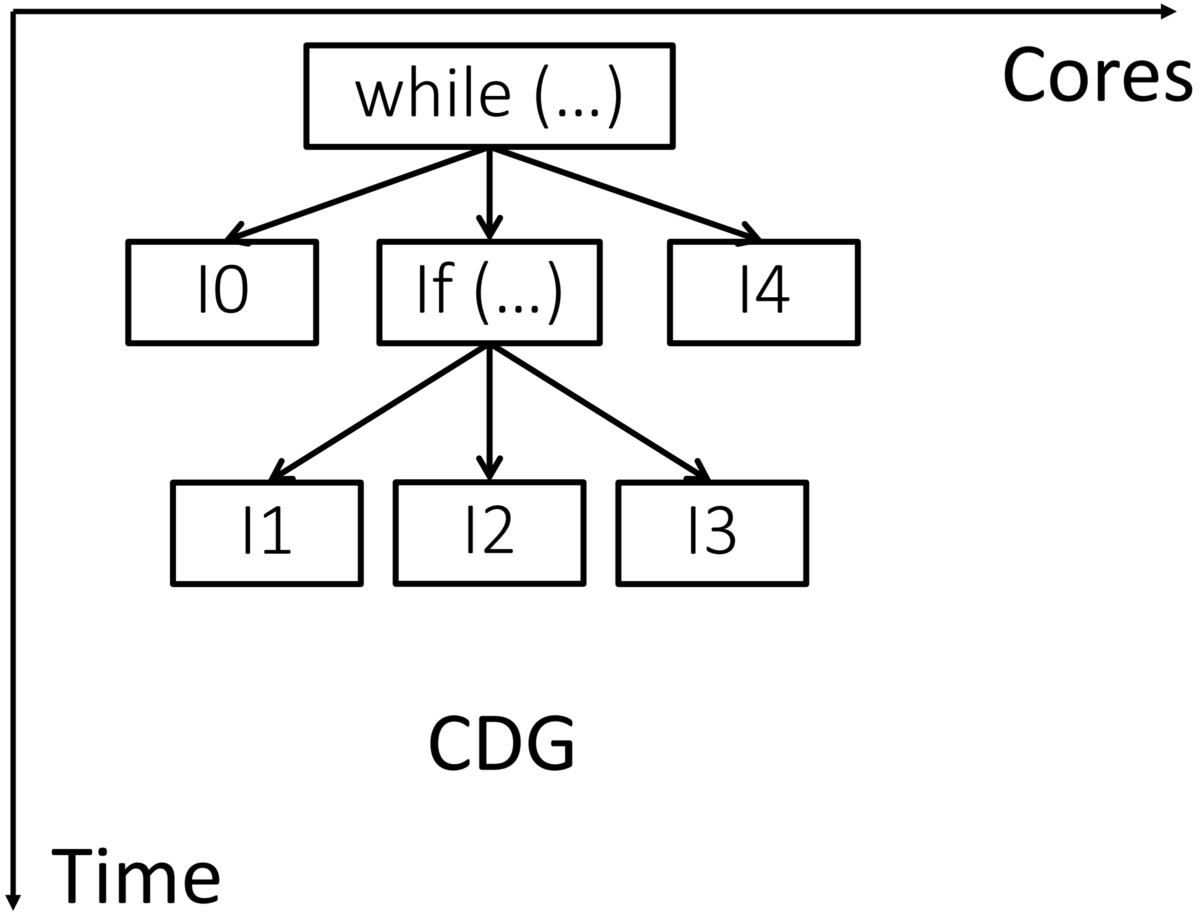
**Sequential program:** fixed order of execution

**Goal:** remove unnecessary order

Useful for parallelism

# Extracting parallelism automatically

```
    while (…)
I0:   …
      if (…){
I1:       …
I2:       …
I3:       …
      }
I4:   …
    }
```




- Assuming
  - no data dependence
  - Infinite cores
- We want to minimize the wall time of our program

# Control dependence graph

- The previous definition of control dependences

A node *X* is control-dependent on another node *Y* if and only if

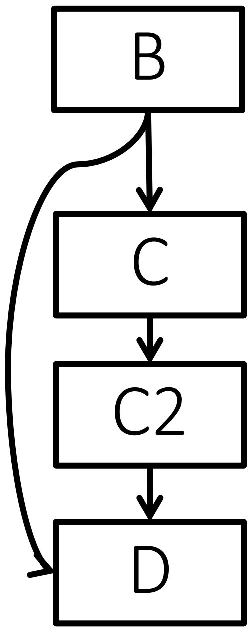
1. There is a path from X to Y such that
   every node in that path other than X is post-dominated by Y
2. X is not strictly post-dominated by Y

- Naïve implementation:
  Iterate over all pair of instructions
  Check conditions 1 and 2 for each pair
      $O(N^2)$
- Can we do better?

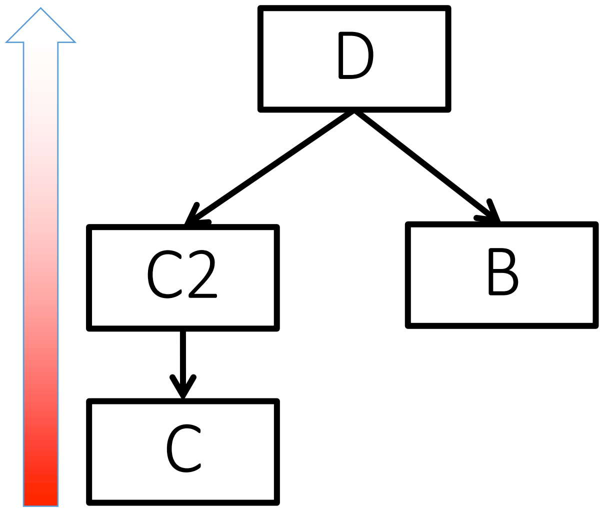
# Control dependence graph: algorithm

A node *Y* control-depends on another node *X* if and only if

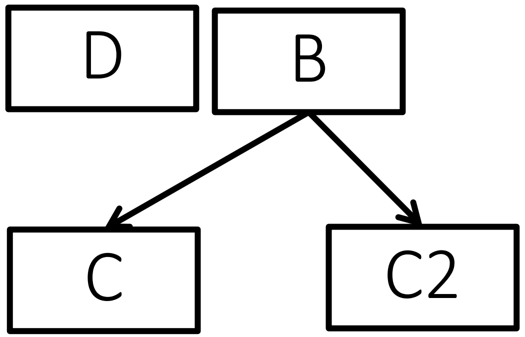
1. There is a path from X to Y such that every node in that path other than X is post-dominated by Y
2. X is not strictly post-dominated by Y




CFG




Immediate post-dominator tree




CDG

(B,C)

(B,C2)

How can we compute the CDG?

# Outline

- Control dependences

- Data dependences

- Introduction to memory alias analysis

# Data dependence

Three types of data dependence (assuming int a,b,c):

- Flow (True) dependence : read-after-write

```
a = c * 10;
b = 2 * a + c;
```

- Anti Dependency: write-after-read

```
a = b* 4+ c;
c = b + 40;
```

- Output Dependence: write-after-write

```
a = b *c ;
a = b + c + 10;
```

# Data dependences

- Gives constraints on parallelism that must be satisfied

- Must be satisfied to have correct program
  - How can we satisfy data dependences?

- Any order that does not violate these dependences is correct!

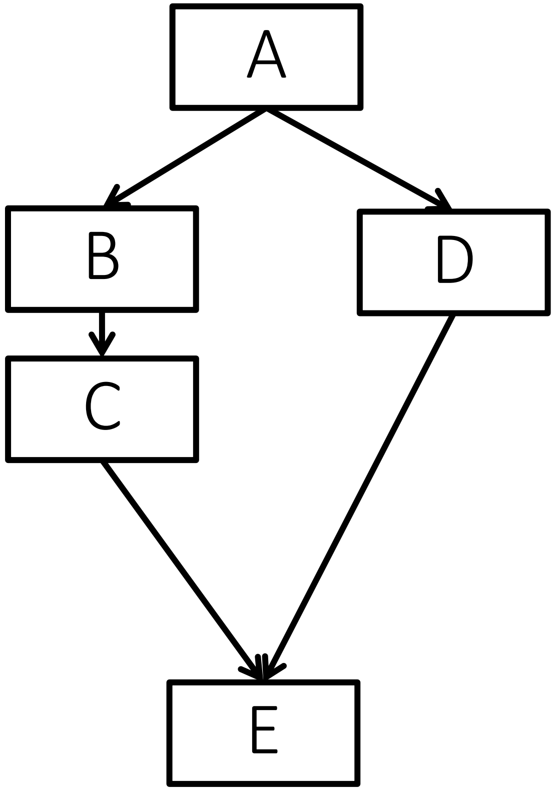
# Data dependence graph (DDG)

- Graph (N, E) where
  - N are instructions
  - Exist an edge (x,y) in E if and only if y is data dependent on x

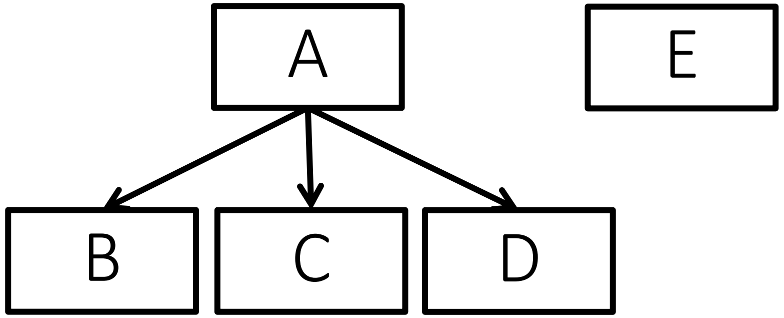
Differences between CDG and DDG

- Granularity
- Structure vs. content

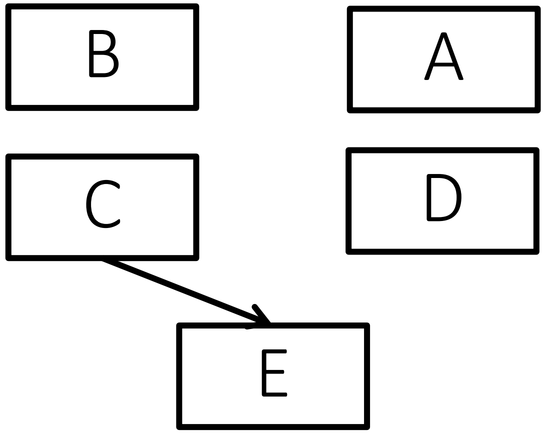
# Dependence example




CFG




CDG




DDG

What are the possible executions that preserve the original semantics of the program?

A B C E   A D E

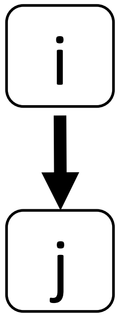
A C B E   A E D   A C E B

# Dependence descriptors

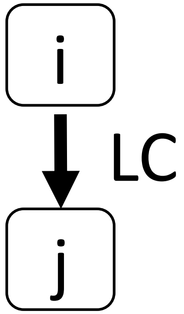
- Data vs. control
- RAW, WAR, WAW
- …

# Loop-carried data dependences

```
while(…){
  i: x = …;
  j: *p = x + 1;
  …
}
```

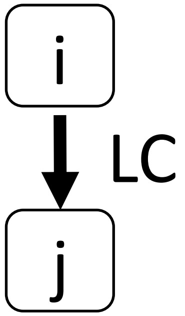



```
while(…){
  j: *p = x + 1;
  i: x = …;
  …
}
```



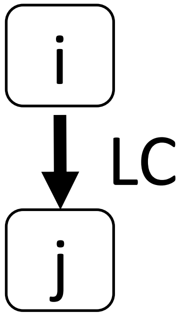
# Loop-carried data dependences

```
while(…){
   j: *p = x + 1;
   i: x = …;
   …
}
```



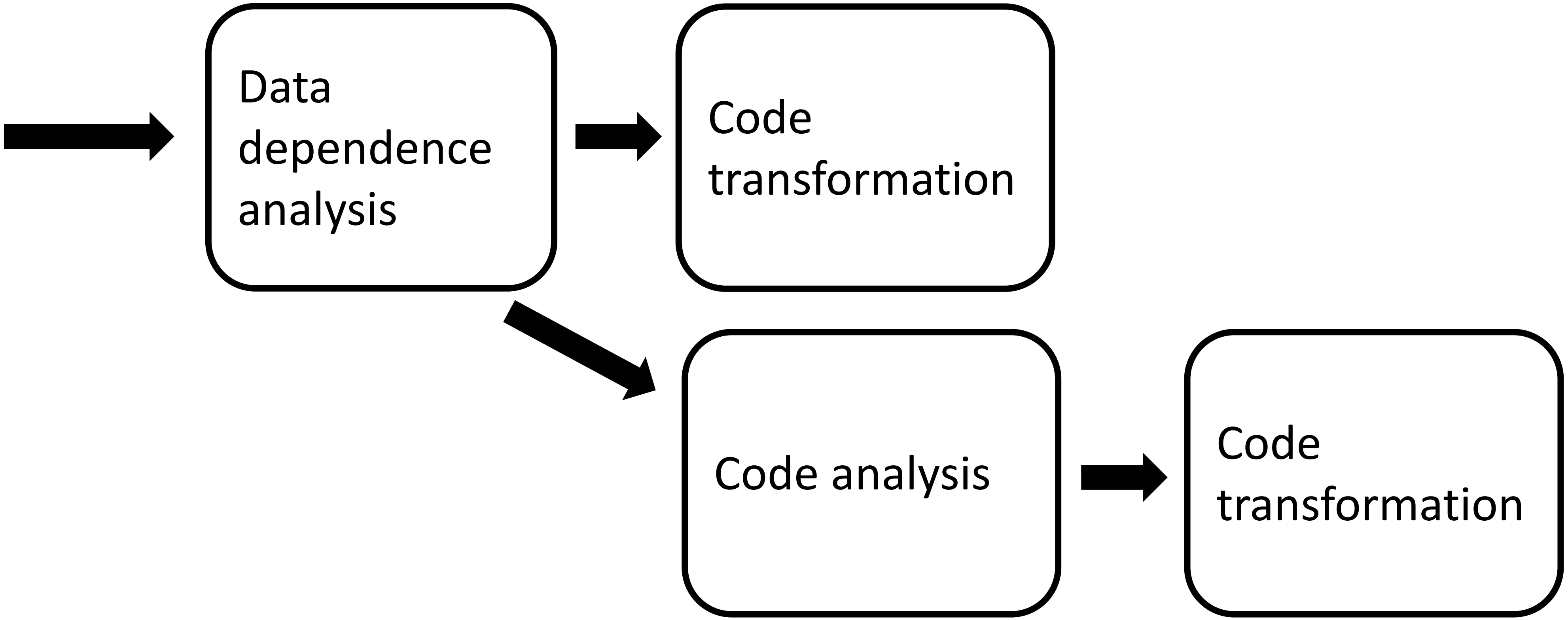

**Distance =1**

```
while(…){
   j: *p = A[i-2] + 1;
   i: A[i] = …;
   k: i++;
}
```




**Distance =2**

# Data dependence analysis and others



# (Variable) Data dependences in LLVM

Any idea?

# (Memory) Data dependences in LLVM

- Memory data dependences are computed by MemoryDependenceAnalysis

```
#include "llvm/Analysis/MemoryDependenceAnalysis.h"
```

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
  AU.addRequired< MemoryDependenceWrapperPass >();
  return;
}
```

- To get the output of the data dependence analysis:

```
MemoryDependenceResults &MD = getAnalysis< MemoryDependenceWrapperPass >().getMemDep();
```

- To get a dependency

```
MemDepResult memInstDeps = MD.getDependency(memInst);
auto memInst2 = memInstDeps.getInst();
```

# Program dependence graph

- Program Dependence Graph = Control Dependence Graph + Data Dependences

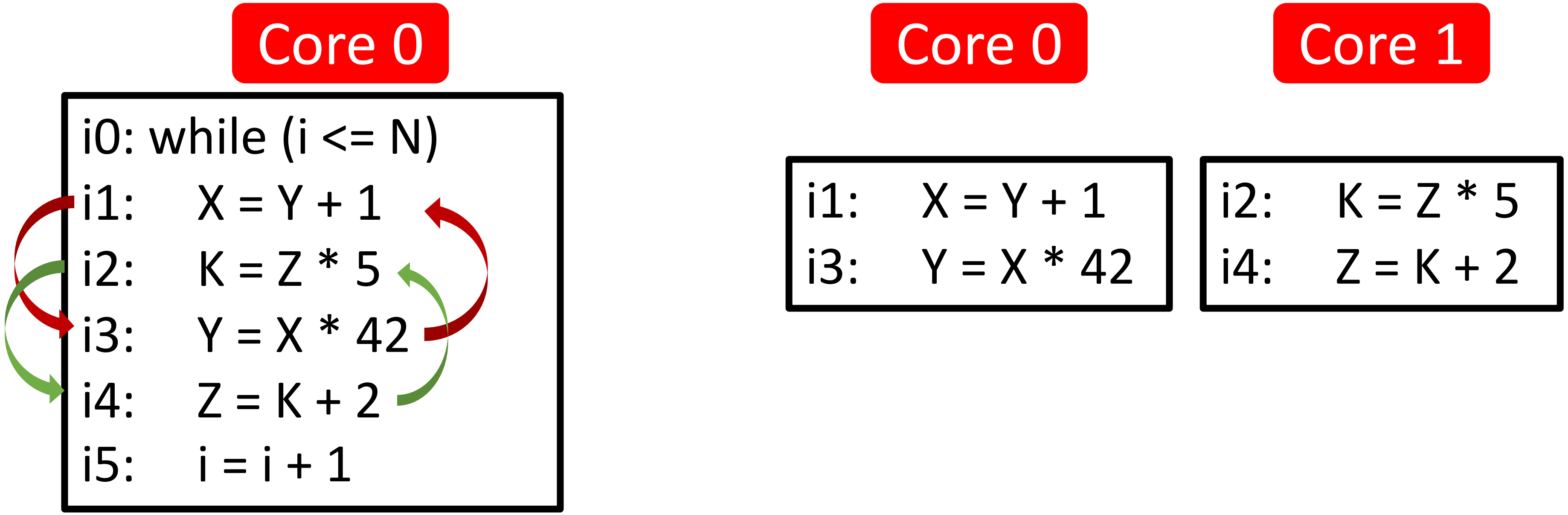
- Facilitates performing most traditional optimizations
  - Constant folding, scalar propagation, common subexpression elimination, code motion, strength reduction

- Requires only single walk over PDG

# Strongly Connected Component (SCC)

Often you need to partition instructions in groups

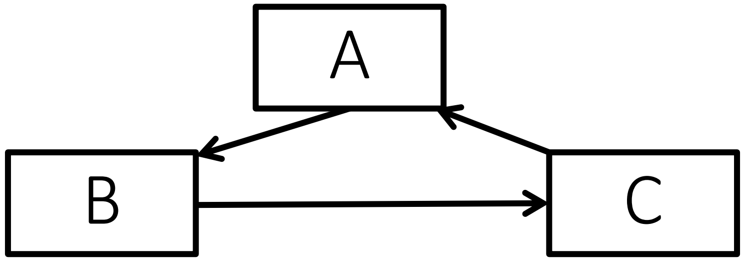
- Where each group is composed of instructions that depend on each other



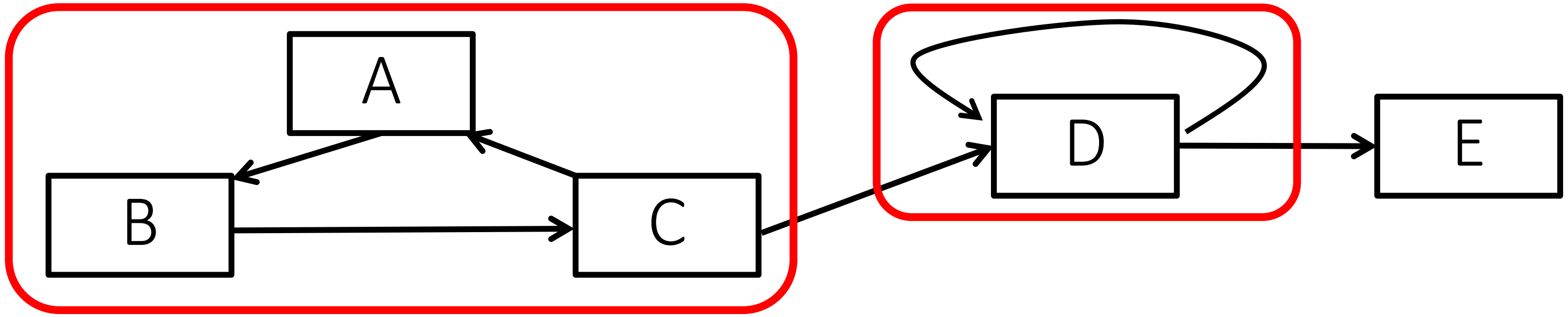

Different colors <-> different cycles in the PDG => different cores

# Strongly Connected Component (SCC)

- A directed graph is strongly connected if there is a path between all pairs of vertices



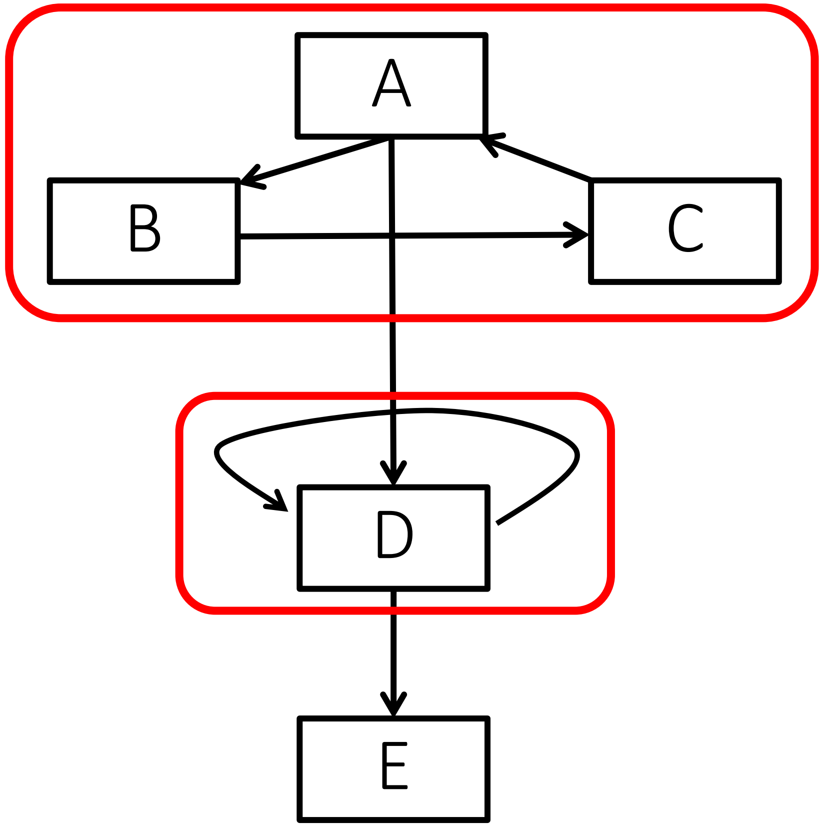

- A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph



# SCCDAG

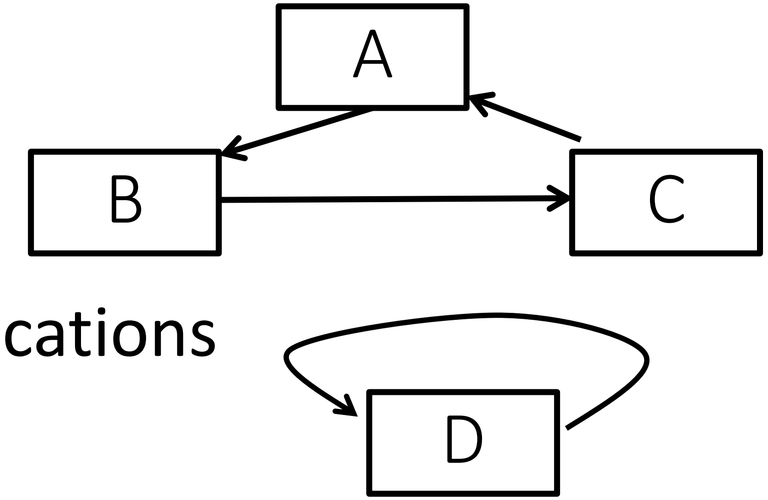
- From the PDG

- To the SCC identifications



# SCCDAG

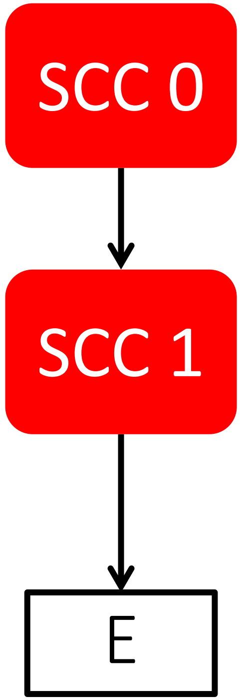
- From the PDG




- To the SCC identifications

- To the SCCDAG

```
i0: while (i <= N)
i1:     X = Y + 1
i2:     K = Z * 5
i3:     Y = X * 42
i4:     Z = K + 2
i5:     i = i + 1
```



# Identify SCCs

- Tarjan's algorithm ← In practice, this is faster
  - It utilizes the property that
    nodes of a strongly connected component form
    a subtree in the DFS spanning tree of the graph
  - Complexity: $O(|N| + |E|)$
- Kosaraju's algorithm
  - It utilizes the property that the transpose graph
    (the same graph with the direction of every edge reversed)
    has the same strongly connected components as the original graph
  - Performs two DFSs on the graph
  - It is similar to the method for finding the topological sorting
  - Complexity: $O(|N| + |E|)$

# Identify SCCs in LLVM (Tarjan’s algorithm)

- **Two template APIs to iterate over SCCs of a graph G:** scc_begin() **and** scc_end()

```
for (auto sccI = scc_begin(pdg); sccI != scc_end(pdg); ++sccI) {
    auto const &scc = *sccI;
}
```

- **These APIs assume the method** getEntryNode() **can be called from the object given as input**
- **The return type of** getEntryNode() **set the type of scc**
  - **E.g., if we have the following method for our pdg:** MyNodeT * getEntryNode ()
  - **Then** scc **is of type** std::vector<MyNodeT *> **and therefore**
  - const std::vector<MyNodeT *> &scc = *sccI;

# Outline

- Control dependences

- Data dependences

- Introduction to memory alias analysis

# Memory alias analysis: the problem

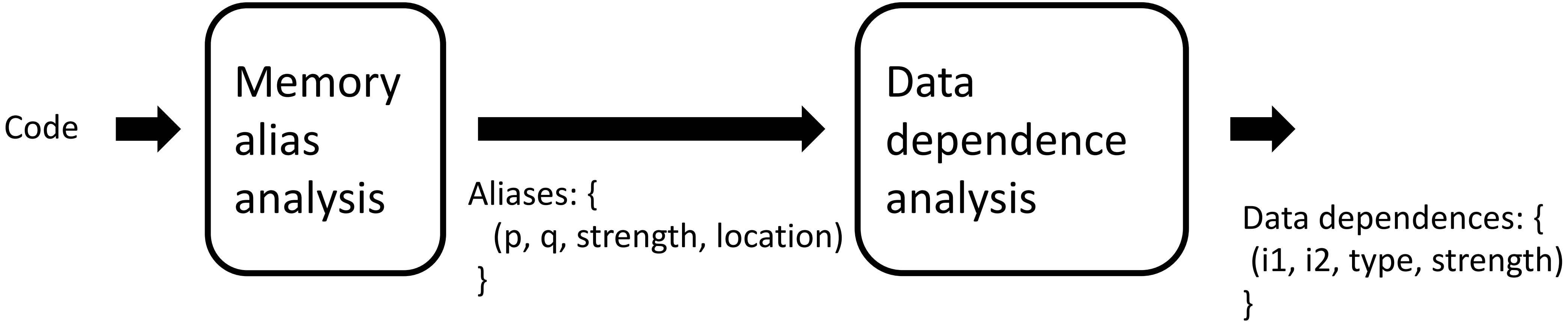
- We want to
  - Execute *j* in parallel with *i* (extracting parallelism)
  - Move *j* before *i* (code scheduling)
- Does *j* depend on *i* ?

```
i: (*p) = varA + 1
j: varB = (*q) * 2
```

```
i: obj1.f = varA + 1
j: varB= obj2.f * 2
```

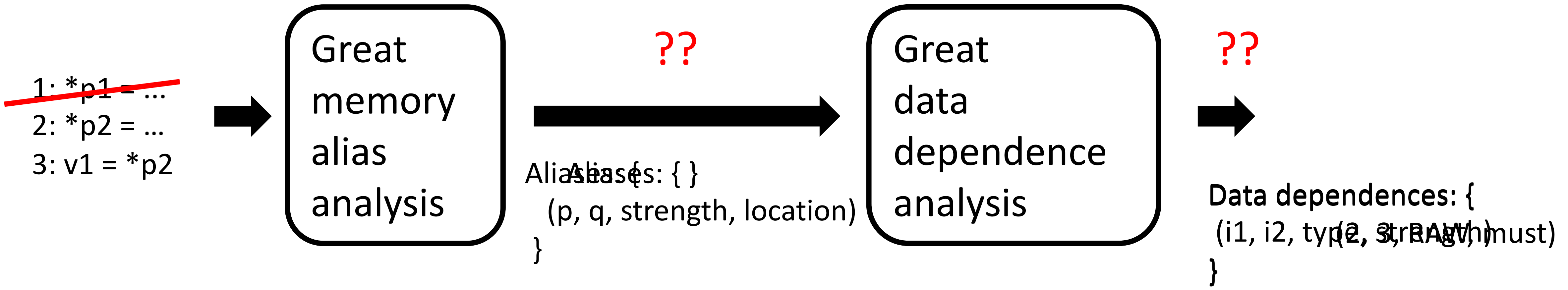
- Do p and q point to the same memory location?
  - Does q alias p?

# Memory alias/data dependence analysis



# Memory alias/data dependence analysis

**Can we optimize the code knowing these dependences?**




Oracle:
p2 and p1 points to different memory locations always

# Memory alias/data dependence analysis

1: *p1 = …
2: *p2 = …
3: v1 = *p2

→ **Not great memory alias analysis** →

Aliases: {
(p1, p2, may, 1)
(p1, p2, may, 2)
(p1, p2, may, 3)
}

→ **Great data dependence analysis** →

Data dependences: {
(2, 3, RAW, must),
(1, 2, WAW, may)
}

Oracle:
p2 and p1 points to different memory locations always

# Memory alias/data dependence analysis

1: *p1 = …
2: *p2 = …
3: v1 = *p2

→ **Not great memory alias analysis** →

Aliases: {
(p1, p2, may, 1)
(p1, p2, may, 2)
(p1, p2, may, 3)
}

→ **Not great data dependence analysis** →

Data dependences: {
(2, 3, RAW, must),
(1, 2, WAW, may),
(1, 3, RAW, may)
}

Oracle:
p2 and p1 points to different memory locations always

Analysis output:
Everything depends on everything else

# Memory alias/data dependence analysis

**Inaccuracies on either memory alias analysis**
**or data dependence analysis**
**leads to “apparent” dependences**

- **More constraints on code transformations**
- **Reduce the aggressiveness of code transformations**
- **Reduce performance obtained**

Oracle:
p2 and p1 points to different memory locations always

Analysis output:
Everything depends on everything else

# Memory alias/data dependence analysis

**Can we optimize the code knowing these dependences?**

~~1: *p1 = …~~
2: *p2 = …
3: v1 = *p2

→ Great memory alias analysis →

Aliases: {
(p1, p2, must, 1)
(p1, p2, must, 2)
(p1, p2, must, 3)
}

→ Great data dependence analysis →

Data dependences: {
(2, 3, RAW, must),
(1, 2, WAW, must)
}

Oracle:
p2 and p1 points to the same memory location always

# Memory alias/data dependence analysis

**We cannot delete instruction 1**

```
1: *p1 = ...
2: *p2 = ...
3: v1 = *p2
```

Not great memory alias analysis

Aliases: {
(p1, p2, may, 1)
(p1, p2, may, 2)
(p1, p2, may, 3)
}

Great data dependence analysis

Data dependences: {
(2, 3, RAW, must),
(1, 2, WAW, may),
}

Oracle:
p2 and p1 points to the same memory location always

# Memory alias/data dependence analysis

**Useless output**

- **Alias analysis:**
  **a pointer may alias to another one**
- **Data dependence analysis:**
  **an instruction may depend on another one**

**… may …**

# Memory alias/data dependence analysis and code analysis/transformation

**Code analysis and transformation**

**that rely on memory alias analysis**
**and/or data dependence analysis**

**must be correct**

**independently with the accuracy of**
**memory alias analysis**
**and/or data dependence analysis**