DFA

Simone Campanoni
simonec@eecs.northwestern.edu

Code analysis and transformation
Data Flow Analysis outline

- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)
Variables and constants

x = 0;
y = x + 1;

Constants

Variable definitions

Variable uses
Now that we know variables, we can talk about how data flows through the code.
Data flows

```c
int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
    y = 0;
    for (int i=0; i <= N; i++){
        x = x + (a * b);
        x = x + b * y;
    }
    return x;
}
```

Data flows from a definition to its uses
Data flow examples

```c
int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
    y = 0;
    for (int i=0; i <= N; i++){
        x = x + (a * b);
        x = x + b*y;
    }
    return x;
}
```

Understanding data flows require understanding the possible sequence of instructions that could be executed at run-time control flows
Control flows

**Control flow:** sequence of instructions in a program that may execute at run-time in that order

*(common simplification: we ignore data values and arithmetic operations)*

```
x = a;
y = x + 1;
x++;
return x + y;
```

```
x = a;
y = x + 1;
if (y > 5){
x--;
} else {
x++;
}
```
How can we automatically identify and represent the control flows?

Let us start by looking at how to iterate over instructions of a function in LLVM
Functions and instructions

What is the instruction that will be executed after `inst`?

The iteration order of instructions isn’t the execution one.
We cannot use iteration order to analyze data flows.
Storing order ≠ executing order

```c
int myF (int a){
    int x = a + 1;
    if (a > 5){
        x++;
    } else {
        x--;
    }
    return x; }
```

When the storing order is chosen (compile time),
the execution order isn’t known
Storing order ≠ executing order

Common pitfall 1:
if instruction i1 has been stored before i2,
then i2 is always executed after i1

Common pitfall 2:
if instruction i1 has been stored before i2,
then i2 can execute after i1
How can we automatically identify and represent the control flows?

We could represent the control flows using a directed graph:
- Node: instruction
- Direct edge: points to the possible next instruction that could be executed at run-time
Representing the control flow of the program

- Most instructions

- Jump instructions

- Branch instructions
Representing the control flow of the program

A graph where nodes are instructions
- Very large
- Lot of straight-line connections
- Can we simplify it?

**Basic block**
Sequence of instructions that is always entered at the beginning and exited at the end
Basic blocks

A basic block is a maximal sequence of instructions such that

- Only the first one can be reached from outside this basic block
- All instructions within are executed consecutively if the first one get executed
  - Only the last instruction can be a branch/jump
  - Only the first instruction can be a label

- Is the storing sequence = execution order in a basic block?
Basic blocks

• Automatically identified
• Algorithm:
  • Code changes trigger the re-identification
  • Increase the compilation time
• Enforced by design
• Instruction exists only within the context of its basic block
• To define a function:
  • you define its basic blocks first
  • Then you define the instructions of each basic block

Inst = F.entryPoint()
B = new BasicBlock()
While (Inst){
    if Inst is Label {
        B = new BasicBlock()
    }
    B.add(Inst)
    if Inst is Branch/Jump{
        B = new BasicBlock()
    }
    Inst = F.nextInst(Inst)
}
Add missing labels
Add explicit jumps
Delete empty basic blocks

What about calls?
  - Program exits
  - Infinite loops in callees
  - Program exits in callees

Delete empty basic blocks
Basic blocks in LLVM

• Every basic block in LLVM must
  • Have a label associated to it
  • Have a “terminator” at the end of it

• The first basic block of LLVM (entry point) cannot have predecessors

```c
6:
  store i32 0, i32* %2, align 4
  br label %14
```
Basic blocks in LLVM

• LLVM organizes “compiler concepts” in containers
  • A module is a container of functions (Module)
  • A function is a container of basic blocks (Function)
  • A basic block is a container of ordered LLVM instructions (BasicBlock)
Basic blocks in LLVM (2)

- LLVM C++ Class "BasicBlock"
- Uses:
  - BasicBlock *b = ... ;
  - Function *f = b.getParent();
  - Module *m = b.getModule();
  - Instruction *i = b.getTerminator();
  - Instruction *i = b.front();
  - size_t b.size();
Basic blocks in LLVM in action

- All function variables are declared at the beginning of the function
- A variable access becomes a memory access
Basic blocks in LLVM in action

```c
function()
{
    int a = 1;  //Sequential instructions
    int b = 2;
    if (b == 2)  //Jump instruction
    {
        ++b;
    }
    int c = 3;  //Sequential instructions
    int d = 4;
    while (a < 5)  //Jump instructions
    {
        ++a;
    }
    int e = 5;
    int f = 6;
}
while.cond:
    %2 = load i32 * %a, align 4
    %cmp1 = icmp slt i32 %2, 5
    br i1 %cmp1, label %while.body, label %while.end
while.body:
    %3 = load i32 * %a, align 4
    %inc2 = add nsw i32 %3, 1
    store i32 %inc2, i32* %a, align 4
    br label %while.cond
while.end:
    store i32 5, i32* %e, align 4
    store i32 6, i32* %f, align 4
    ret void
```
How can we automatically identify and represent the control flows?

We could represent the control flows using a directed graph:

- **Node**: instruction
  - **Basic block**

- **Direct edge**: points to the possible next instruction that could be executed at run-time
Control Flow Graph (CFG)

- A CFG is a graph $G = \langle \text{Nodes, Edges} \rangle$
- Nodes: Basic blocks
- Edges: $(x, y) \in \text{Edges}$ iff
  
  first instruction of basic block $y$ ($I_y$) **may** be executed just after the last instruction of the basic block $x$ ($I_x$)
Control Flow Graph (CFG)

- Entry node: block with the first instruction of the function
- Exit nodes: blocks with the return instruction
  - Some compilers make a single exit node by adding a special node
CFG example

```
function()
{
    int a = 1;  // Sequential instructions
    int b = 2;  // ---------------

    if (b == 2)  // Jump instruction
    {
        ++b;  // Jump target
    }

    int c = 3;  // Sequential instructions
    int d = 4;  // ---------------

    while (a < 5)  // Jump instruction and jump target
    {
        ++a;  // Jump target
    }

    int e = 5;  // Sequential instructions
    int f = 6;  // ---------------
}
```
CFG in LLVM

Differences?

Bitcode generation

opt -view-cfg
F.viewCFG();
Navigating the instructions within a basic block

```cpp
auto nextInstruction = i->getNextNode();
auto prevInstruction = i->getPrevNode();
```
Navigating the CFG in LLVM: from a basic block to another

Successors of a basic block

```cpp
for (auto succBB : successors(bb)){
```

Predecessors of a basic block

```cpp
for (auto predBB : predecessors(bb)){
```
Navigating the CFG in LLVM: From an instruction to its successors

Let’s say we want to iterate over the successors of i so from i to j and k

How can we do it?

```cpp
for (auto succBB : successors(bb)) {
    // code
}
```
Navigating the CFG in LLVM:
From an instruction to its successors

```cpp
for (auto &b : F)
  auto i = b.getTerminator();
  errs() << *i << "\n";
  for (auto succBB : successors(b))
    auto firstInstOfSuccBB = succBB->front();
```

H0/tests

Output of the LLVM pass of the previous slide:
Code transformation example: constant propagation

int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
    y = 0;
    for (int i=0; i <= N; i++){
        x = x + a * b; \[\text{Replace a variable use with a constant}\]
        x = x + b * y; \[\text{while preserving}\]
        \[\text{the original code semantics}\]
    }
    return x;
}
Code transformation example: constant propagation

```c
int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
    y = 0;
    for (int i=0; i <= N; i++){
        x = x + (a * b);
        x = x + b*y;
    }
    return x;
}
```

Replace a variable use with a constant while preserving the original code semantics.
Code transformation example: constant propagation

int sumcalc(int a, int b, int N) {
    int x, y;
    x = 0;
    y = 0;
    for (int i = 0; i <= N; i++) {
        x = x + (a * b);
        x = x + b * 0;
    }
    return x;
}
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
The need for DFAs

• We constantly need to improve programs (e.g., speed, energy efficiency, memory requirements)
• We constantly need to identify opportunities
• After having found an opportunity (e.g., propagating constants), you need to ask yourself:
  • What do I need to know to take advantage of this opportunity? (e.g., I need to know the possible values a given variable might have at a given point in the program)
  • How can I automatically compute this information? Often the solution relies on understanding how data flows through the code. This is often done by designing ad-hoc DFAs
Let us go deeper in the need for data flow analysis for code transformation

Let us introduce an actual code transformation implemented by all compilers: constant propagation
Transformation: constant propagation
Analysis: reaching definition DFA

• Opportunity: this code is “better” compared to this

Which information do I need to know if it is safe to replace $b$ with 2

Among all possible run time control flows, what are the latest definitions of $b$?

What are the possible values $b$ can have at run time?

Which information do I need to know if it is safe to replace $b$ with 2

Among all possible run time control flows, what are the latest definitions of $b$?

What are the possible values $b$ can have at run time?
Constant propagation

- Find an instruction $i$ that defines a variable with a constant expression
  
  $\text{Instruction } i: \ b = \text{CONSTANT\_EXPRESSION}$

- Replace an use of $b$ in an instruction $j$ with that $\text{CONSTANT\_EXPRESSION}$ if
  
  - All control flows to $j$ includes $i$
  
  - There are no intervening definition of that variable

![Diagram showing constant propagation](image-url)
Constant propagation: code example

```c
int sumcalc (int a, int b, int N){
    int x,y;
    x = 0;
    y = 0;
    if (a > b){
        x = x + N;
    }
    if (b > N){  return y;}
    return x;
}
```

**Data-flow analysis** is a collection of techniques for compile-time reasoning about the run-time values.

We need to analyze the “data-flows” of a program and represent them explicitly.
But constant propagation (CP) has been done already ...

• CP has been already designed and implemented

• Why should we study it? Why don’t we design and implement all possible transformations and analyses in a compiler and move on?

• It is always possible to invent new/better transformations

Full employment theorem for compiler writers
New transformations and analyses

• New transformations (often) need to understand specific and new code properties related to how data might change through the code
  • So we need to know how to design a new data flow analysis that identifies these new code properties

• Generic recipe
  Data flow analysis (DFA): traverse the CFGs collecting information about what may happen at run time (Conservative approximation)
  Transformation: Modify the code based on the result of data flow analysis (Correctness guaranteed by the conservative approximation of DFA)
New transformations and analyses

• Generic recipe

**Data flow analysis (DFA):**

traverse the CFGs collecting information about what may happen at run time (Conservative approximation)

**Transformation:**

Modify the code based on the result of data flow analysis (Correctness guaranteed by the conservative approximation of DFA)

Among all possible run time control flows, what are the latest definitions of \( b \)?

What are the possible values \( b \) can have at run time?

Data flow value
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
Concepts

• Static and dynamic control flows

• Data flow abstraction

• Data flow values

• Transfer functions

• GEN, KILL, IN, OUT sets
Static program vs. dynamic execution

- **Static**: Finite program
- **Dynamic**: Can have infinitely many possible control flows
- **Data flow analysis abstraction**: For each point in a program: combine information about all possible run-time instances of the same program point.

What are the possible values of $b$?

Data flow analysis (DFA): traverse the CFGs collecting information about what may happen at run time (Conservative approximation)
Example of data-flow questions

• What are the possible values of \( b \) just before an instruction “\( ... = b \)”?
• Which instruction defines the value used in “\( ... = b \)”?
Example of data-flow questions

• What are the possible values of b just before an instruction “... = b”?
• Which instruction defines the value used in “... = b”?
• Has the expression “a * b” been computed before another instruction? (“... = a * b”)
• What are the instructions that might read the value produced by an instruction “b = ...”?
• What are the instructions that will (must) read the value produced by an instruction “b = ...”?
• ...
Data-flow expressed in CFG

Data-flow value:
set of all possible program states that can be observed at a given program point
e.g., all definitions in the program that might have been executed before that point

Data-flow analysis
computes IN and OUT sets by computing the DFA-specific transfer functions

IN={ }
\{ x=0 \} = OUT

int x, y
x = 0
y = 0
If (a > b)

x = x + N

If (b > N)

return y

return x

{ }
Transfer functions

• Let $i$ be an instruction: $\text{IN}[i]$ and $\text{OUT}[i]$ are the set of data-flow values before and after the instruction $i$ of a program.

• A transfer function $fs$ relates the data-flow values before and after an instruction $i$.

• In a forward data-flow problem
  \[ \text{OUT}[i] = fs(\text{IN}[i]) \]

• In a backward data-flow problem
  \[ \text{IN}[i] = fs(\text{OUT}[i]) \]

$fs$ is DFA-specific.
Transfer function internals: \( Y[i] = fs(X[i]) \)

- It relies on information that reaches \( i \)

- It transforms such information to propagate the result to the rest of the CFG

  \[ \text{GEN}[i] = \text{data flow value added by } i \]
  \[ \text{KILL}[i] = \text{data flow value removed because of } i \]

- To do so, it relies on information specific to \( i \)
  - Encoded in \( \text{GEN}[i], \text{KILL}[i] \)
    - \( fs \) uses \( \text{GEN}[i] \) and \( \text{KILL}[i] \) to compute its output

- \( \text{GEN}[i] \) and \( \text{KILL}[i] \) are DFA-specific and (typically) data/control flow independent!

```plaintext
int x, y
x = 0
y = 0
If (a > b)
  \{ x=0 \} = \text{OUT}
```
DFA steps

1) Define the DFA-specific sets GEN[i] and KILL[i], for all i and without looking at the control flows

2) Implement the DFA-specific transfer function $fs$

3) Compute all $IN[i]$ and $OUT[i]$ following a DFA-generic algorithm
   
   $OUT[i] = fs ( IN[i] )$
   $IN[i] = fs ( OUT[i] )$

Compilers have a data flow framework to help developing new DFAs (we will not rely on such framework for this class)
Data Flow Analysis outline

- Concepts needed by most code analyses
- Why do we need DFA? (opportunities)
- Introduction to DFA (concepts)
- A DFA example: reaching definitions (concept application)
- Implementation of DFA (actual implementation)
Before introducing the reaching definition DFA, let us go back to the previous example to formalize new terminology.
Optimization example: constant propagation

int sumcalc (int a, int b, int N) {
    int x, y;
    x = 0;
    y = 0;
    if (a > b) {
        x = x + N;
    }
    if (b > N) {
        return y;
    }
    return x;
}

IN[return y] = \{y=0\}
IN[return x] = \{x=0, x = x + N\}

Information needed just before an instruction $i$:
what are the definitions that might execute before $i$?
reach

if (b > N) {
    return 0;
}
Let us define the concept of “reaching” more formally
Data-flow example: reaching definitions

- A definition \( D \) reaches a program point \( X \) if there is a control flow from \( D \) to \( X \) such that the variable defined by \( D \) is not redefined along that path.

\[
\begin{align*}
D &: \ v = 0 \\
J &: \ \text{call printf(...)}
\end{align*}
\]

\[
\begin{align*}
X &: \ ... = v ...
\end{align*}
\]

\[
\begin{align*}
\text{GEN}[D] &= \{D\} \\
\text{IN}[X] &= \{D\}
\end{align*}
\]
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

$$D: v = 0$$
$$J: \text{call printf(...)}$$
$$X: \ldots = v \ldots$$

$D$ reaches $X$

$IN[X] = \{D\}$

$GEN[D] = \{D\}$
Data-flow example: reaching definitions

- A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

```
D: v = 0
...
J: v = v + n
...
X: ... = v ...
```

- $D$ does not reach $X$.
- $D$ is not in $\text{IN}[X]$.
- $\text{GEN}[D] = \{D\}$, $\text{KILL}[D] = \{J\}$.
- $\text{GEN}[i] = \text{data flow value added by } i$.
- $\text{KILL}[i] = \text{data flow value removed because of } i$. 
Data-flow example: reaching definitions

- A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

```
...  
D: v = 0  
...  
J: v = v + n  
...  
X: ... = v ...
```

- $D$ reaches $X$
  - $IN[X] = \{J\}$
  - $GEN[D] = \{D\}$
  - $KILL[D] = \{J\}$

- $J$ reaches $X$
  - $KILL[J] = \{D\}$
  - $GEN[J] = \{J\}$

- $KILL[J] = \{D\}$
Data-flow example: reaching definitions

• A definition $D$ reaches a program point $X$ if there is a control flow from $D$ to $X$ such that $D$ is not killed along that path.

• The reaching definition data-flow problem for a flow graph is to compute all definitions that reach an instruction $i$ (i.e., $IN[i]$, $OUT[i]$) for all $i$ in that graph.
Computing INs and OUTs

\[ \text{IN}[i] = \text{OUT}[p] \]

\[ \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] \setminus \text{KILL}[i]) \]

- Forward or backward?
  \[ \text{OUT}[i] = f_s(\text{IN}[i]) \]

- GEN\[i\] = what \(i\) generates
- KILL\[i\] = what \(i\) kills (invalidates)

- \(f_s\) within a basic block?
- Let \(i\) be an instruction and \(p\) be its only predecessor

\[ \text{IN}[i] = \text{OUT}[p] \]

\[ \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] \setminus \text{KILL}[i]) \]

Local reaching definitions
Data-flow example: reaching definitions

• A definition $d$ reaches a program point $X$ if there is a path from $d$ to $X$ such that $d$ is not killed along that path.

• The data-flow problem for a flow graph is to compute $\text{IN}[i]$ and $\text{OUT}[i]$ for all $i$ in that graph.

$\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]$  
$\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])$

Should 1 be in $\text{IN}[5]$?
Data Flow Analysis outline

• Concepts needed by most code analyses

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
• So far, we have defined **data-flow equations** (i.e., IN and OUT equations)

• How can we actually compute them?

• Main problem:
  ➢ input of equation IN depends on output of equation OUT
    \[
    \text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]
    \]
  ➢ Output of equation OUT depends on input of equation IN
    \[
    \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])
    \]
We break all possible dependence cycles by iteratively computing all IN and OUT sets until a fixed point is reached.
Steps for iterative algorithm

• Compute GEN and KILL sets for all instructions without using the CFG
  • GEN and KILL sets will not change anymore

• Compute IN and OUT sets with an iterative algorithm
  do{
    Compute IN and OUT sets for all instructions
  } while (any IN or OUT set changes from the previous iteration)
Iterative algorithm for reaching definitions

• Given GEN[i], KILL[i] for all instructions i, we compute IN[i] and OUT[i] for all i

for (each instruction i)  IN[i] = OUT[i] = { };

do {
    for (each instruction i) {
        IN[i] = ∪p a predecessor of i OUT[p];
        OUT[i] = GEN[i] ∪ (IN[i] – KILL[i]);
    }
} while (changes to any OUT occur)
Reaching definition in action

| GEN[0] = {} | KILL[0] = {} |
| GEN[1] = {1} | KILL[1] = {4} |

Why do we need to reach a fixed point?

\[
\text{IN}[i] = \bigcup_{p \text{ a predecessor}} \text{OUT}[p]
\]

\[
\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i])
\]

Done?

0: int x,y
1: x = 0
2: y = 0
3: If (a > b)
4: x = x + N
5: If (b > N)
Now that you know reaching definition

• It’s time for the homework H1
• What we learned was for forward data-flow analysis
  \[ \text{OUT}(s) = \text{fs}(\text{IN}(s)) \]

  for (each instruction \(i\)) \(\text{IN}[i] = \text{OUT}[i] = \{\}\);  
do {
    for (each instruction \(i\)) {
      \(\text{IN}[i] = \bigcup_{p \text{ a predecessor of } i} \text{OUT}[p]\);
      \(\text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] \setminus \text{KILL}[i])\);
    }
  } while (changes to any \(\text{OUT}\) occur)

• What about backward data-flow analysis?
  \[ \text{IN}(s) = \text{fs}(\text{OUT}(s)) \]
Forward DFA

for (each instruction $i$) \[ \text{IN}[i] = \text{OUT}[i] = \{ \}; \]
do {
    for (each instruction $i$) {
        \[ \text{IN}[i] = \text{fs}_{p \text{ a predecessor of } i}(\text{OUT}[p]) \]
        \[ \text{OUT}[i] = \text{fs}(\text{IN}[i]) \]
    }
} while (changes to any OUT occur)
Backward DFA

for (each instruction $i$) $\text{IN}[i] = \text{OUT}[i] = \{ \}$;
do {
  for (each instruction $i$) {
    $\text{OUT}[i] = \text{fs}_{s \text{ a successor of } i}(\text{IN}[s])$
    $\text{IN}[i] = \text{fs}(\text{OUT}[i])$
  }
} while (changes to any $\text{IN}$ occur)
Now that we know DFAs and how to compute them,

let us look at how to reduce the computation time to compute them
Implementation aspects

for (each instruction $i$)  $IN[i] = OUT[i] = \{ \}$;

do {
    for (each instruction $i$) {
        $IN[i] = \bigcup_{\text{a predecessor of } i} OUT[p]$;
        $OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$;
    }
} while (changes to any $OUT$ occur)

• Memory representation of data flow values
• Operations performed on them
• What is an element in a set?
Can we optimize the analysis?

for (each instruction $i$) \( \text{IN}[i] = \text{OUT}[i] = \{ \} \);

do {
    for (each instruction $i$) {
        \( \text{IN}[i] = \bigcup p \text{ a predecessor of } i \ \text{OUT}[p] \);
        \( \text{OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] - \text{KILL}[i]) \);
    }
} while (changes to any OUT occur)

... that’s a lot of iterations repeated for each while iteration

Is this always necessary?
for (each basic block $B$) \( \text{IN}[B] = \text{OUT}[B] = \{ \} \);

do {
    for (each basic block $B$) {
        \( \text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P] \);
        \( \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]) \);
    }
}while (changes to any OUT occur)

Optimization 1: basic blocks

Contains all definitions in block $B$ that are visible immediately after $B$

Contains all definitions killed by instructions in block $B$

Suggestion: if you are going to implement these optimizations, then either
• skip this one or
• keep it to be the last one
Optimization 2: bit-set

for (each basic block B) \( \text{IN}[B] = \text{OUT}[B] = \{ \} \); do {
  for (each basic block B) {
    \( \text{IN}[B] = \bigcup_{\text{a predecessor of } B} \text{OUT}[P] \); 
    \( \text{OUT}[B] = \text{GEN}[B] \bigcup (\text{IN}[B] \bigcap \text{KILL}[B]) \);
  }
} while (changes to any OUT occur)
Optimization 2: bit-sets

- Assign a bit to each element that might be in the set
  - Union: bitwise OR
  - Intersection: bitwise AND
  - Subtraction: bitwise NEGATE and AND

- Fast implementation
  - 64 elements packed to each word on today’s commodity processors
  - AND and OR are single machine code instructions (single cycle latency)
Optimization 3: work list

for (each basic block B) \( \text{IN}[B] = \text{OUT}[B] = \{ \} \);

do {

for (each basic block B) {

\[ \text{IN}[B] = \bigcup_{\text{a predecessor of } B} \text{OUT}[P]; \]

\[ \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]); \]

}

} while (changes to any OUT occur)
Optimization 3: work list

OUT[ENTRY] = { };
for (each basic block B other than ENTRY) OUT[B] = { };
workList = all basic blocks
while (workList isn’t empty)
   B = pick and remove a block from workList
   oldOUT = OUT[B]
   IN[B] = ∪ₚ a predecessor of B OUT[ᵢ]
   if (oldOut != OUT[B]) workList = workList U {all successors of B}
Optimization 4: block order

\[
\text{OUT}[\text{ENTRY}] = \{ \};
\]
for (each basic block \( B \) other than \( \text{ENTRY} \)) \( \text{OUT}[B] = \{ \} \);
workList = all basic blocks
while (workList isn’t empty)

\[ B = \text{pick and remove a block from workList} \]
oldOUT = OUT\([B]\)
\[ \text{IN}[B] = \bigcup_{p \text{ a predecessor of } B} \text{OUT}[p]; \]
\[ \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] – \text{KILL}[B]); \]
if (oldOut != OUT\([B]\)) workList = workList U \{ \text{all successors of } B \} \]
}
Food for thought

• Correctness: is the answer ALWAYS correct?
• Meaning: what is exactly the meaning of the answer?
• Precision: how good is the answer?
• Convergence:
  • Will the analysis ALWAYS terminate?
  • Under what conditions does the iterative algorithm converge?
• Speed: how long does it take to converge?