• **Motivation:** language-independent and machine-independent representation of control flow in programs used in high-level and low-level code optimizers. The flow graph data structure lends itself to use of several important algorithms from graph theory.
Basic Blocks

- A straight line sequence of code
  - a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them.
- First instruction of basic block can be:
  - entry point of routine
  - target of branch
  - instruction immediately following branch - *leaders*
- extended basic block is maximal sequence that contains no join nodes other than first node.
  - Join node: if it has more than one predecessor
  - Branch node: if it has more than one successor
Control Flow Graph: Definition

A control flow graph $CFG = (N_c; E_c; T_c)$ consists of

- $N_c$, a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e. a basic block.
- $E_c \subseteq N_c \times N_c \times \text{Labels}$, a set of labeled edges.
- $T_c$, a node type mapping. $T_c(n)$ identifies the type of node $n$ as one of: START, STOP, OTHER.

We assume that $CFG$ contains a unique START node and a unique STOP node, and that for any node $N$ in $CFG$, there exist directed paths from START to $N$ and from $N$ to STOP.
Dominators: Definition

Node $V$ dominates another node $W \neq V$ if and only if every directed path from $START$ to $W$ in $CFG$ contains $V$.

Define $dom(W) = \{ V | V \text{ dominates } W \}$, the set of dominators of node $W$.

Consider any simple path from $START$ to $W$ containing $W$'s dominators in the order $V_1, \ldots, V_k$. Then all simple paths from $START$ to $W$ must contain $W$'s dominators in the same order. The element closest to $W$, $V_k = idom(W)$, is called the immediate dominator of $W$. 
Control Flow Graphs

- From code can construct “flowchart”
- from flowchart to basic blocks
- edges come from the control flow information
Control Flow Analysis

• Use *dominators* to discover loops and simply note loops found for use in optimization

• alternate approach- *interval analysis*: Analyze overall structure of routine and decompose it into nested regions called intervals
  – nesting structure forms tree, useful in structuring and speeding up data flow analysis
Dominators: Definition (Contd.)

The $idom$ relation can be represented as a directed tree with root = $START$, and $parent(W) = idom(W)$. 
Postdominators: Definition

- Node $W$ postdominates another node $V \neq W$ if and only if every directed path from $V$ to $STOP$ in $CFG$ contains $W$.
- Define $pdom(V) = \{ W | W$ postdominates $V \}$, the set of postdominators of node $V$.
- Consider any simple path from $V$ to $STOP$ containing $V$'s postdominators in the order $W_1, \ldots, W_k$. Then all simple paths from $V$ to $STOP$ must contain $V$'s postdominators in the same order. The element closest to $V$, $W_1 = ipdom(V)$, is called the immediate postdominator of $V$.
- The $ipdom$ relation can be represented as a directed tree with root = is $STOP$ and $parent(V) = ipdom(V)$.
Example: Dominator and Postdominator Trees
Algorithms for computing Dominator/Postdominator

- [Purdom and Moore, 1972] $O(N \times E)$ execution time

- [Lengauer and Tarjan, 1979]
  Simple version: $O(E \times \log N)$ execution time
  Sophisticated version: $O(E \times \alpha(E,N))$ execution time

- [Harel, 1985] $O(N + E)$ execution time
The loop nesting structure of a CFG is revealed by its *interval structure*:

- Edge $e = (x, h, l)$ in CFG is called a *back edge* if $h \in \text{dom}(x)$; $h$ is called a *header* node, and $x$ is called a *latch node*.

- The strongly connected region defined by back edge $e = (x, h, l)$ is $\text{STR}(e)$, which consists of the nodes and edges belonging to all paths from node $h$ to node $x$, along with the back edge $(x, h, l)$.

- The *interval with header* $h$, $I(h)$, is defined as the union of $\text{STR}(e)$ over all back edges $e$ targeted to header node $h$. 
Loop Nesting Structure of a Control Flow Graph (Contd.)

• Interval nesting is defined by the subgraph relationship. \( I(h_1) \) is a subinterval of \( I(h_2) \) if \( I(h_1) \) is a subgraph of \( I(h_2) \).

• The interval nesting relation can then be represented by a unique interval nesting tree (or forest of trees).

• It is convenient to add a pseudo-edge from \textit{STOP} to \textit{START} to make \textit{START} a header node with \( I(START) = \) entire CFG, and thus force the interval relation to be a single tree rather
Reducible Control Flow Graphs

- A CFG is *reducible* if and only if the directed graph obtained by removing all back-edges is *acyclic*. This graph is referred to as the *forward control flow graph*. 
Example: Interval Structure for a Reducible CFG

INTERVALS IN A CONTROL FLOW GRAPH
CS297- B. Narahari

* Back edge
Example: Interval Structure for an Irreducible CFG

INTERVALS IN A CONTROL FLOW GRAPH

* Back edge

INTERVAL NESTING TREE

CS297- B. Narahari
Motivation: identify only the essential control and data dependences which need to be obeyed by transformations for code optimization.

**Program Dependence Graph (PDG)** consists of
1. Set of nodes, as in the CFG
2. Control dependence edges
3. Data dependence edges

Together, the control and data dependence edges dictate whether or not a proposed code transformation is legal.
Control Dependence Analysis

We want to capture two related ideas with control dependence analysis of a CFG:

1. Node $Y$ should be control dependent on node $X$ if node $X$ evaluates a predicate (conditional branch) which can control whether node $Y$ will subsequently be executed or not. This idea is useful for determining whether node $Y$ needs to wait for node $X$ to complete, even though they have no data dependences.
2. Two nodes, $Y$ and $Z$, should be identified as having identical control conditions if in every run of the program, node $Y$ is executed if and only if node $Z$ is executed. This idea is useful for determining whether nodes $Y$ and $Z$ can be made adjacent and executed concurrently, even though they may be far apart in the CFG.
Control Dependence: Definition

[Ferrante et al, 1987]

Node $Y$ is control dependent on node $X$ with label $L$ in $CFG$ if and only if

1. there exists a nonnull path $X \rightarrow Y$, starting with the edge labeled $L$, such that $Y$ post-dominates every node, $W$, strictly between $X$ and $Y$ in the path, and

2. $Y$ does not post-dominate $X$

$Y$ is control dependent on $X$ only if $X$ can directly affect whether $Y$ is executed or not; indirect control dependence can be defined as the transitive closure of control dependence.
Example: acyclic CFG and its Control Dependence Graph

**CONTROL FLOW GRAPH**

**CONTROL DEPENDENCE GRAPH**

**POSTDOMINATOR TREE**
Example: CFG with multi-way branch and its CDG

CONTROL FLOW GRAPH

CONTROL DEPENDENCE GRAPH

POSTDOMINATOR TREE
Algorithm for Computing Control Dependence

Given node $X$ and branch label $L$, all control dependence successors can be enumerated as follows:

1. $Z \leftarrow$ CFG successor of node $X$ with label $L$

2. while $Z \neq ipdom(X)$ do

   (a) /* $Z$ is control dependent on $X$ with label $L$ */ process $Z$ as desired */

   (b) $Z \leftarrow ipdom(Z)$

end while
Example: cyclic CFG and its Control Dependence Graph

CONTROL FLOW GRAPH

START

1

2

3

4

STOP

T

F

T

F

T

U

U

F

POSTDOMINATOR TREE

START

STOP

3

1

4

2

CONTROL DEPENDENCE GRAPH

START

1

2

4

3

T

T

F
Example: another cyclic CFG and its CDG

CONTROL FLOW GRAPH

POST-DOMINATOR TREE

CONTROL DEPENDENCE GRAPH

START

ENTRY

STOP

ENTRY

1

2

3

F

F

F

T

T

T

CS297- B. Narahari
Properties of Control Dependence

- CDG is a *tree* $\implies$ CFG is *structured*
- CDG is *acyclic* $\implies$ CFG is *acyclic*
- CDG is *cyclic* $\implies$ CFG is *cyclic*

The *control conditions of node* $Y$ *is the set*,

$$CC(Y) = \{(X,L) | Y \text{ is control dependent on } X \text{ with label } L\}$$

Two nodes, $A$ and $B$, are said to be *identically control dependent* if and only if they have the same set of control conditions i.e. $CC(A) = CC(B)$
Data Dependence Analysis

If two operations have potentially interfering data accesses, data dependence analysis is necessary for determining whether or not an interference actually exists. If there is no interference, it may be possible to reorder the operations or execute them concurrently.

The data accesses examined for data dependence analysis may arise from array variables, scalar variables, procedure parameters, pointer dereferences, etc. in the original source program.

Data dependence analysis is conservative, in that it may state that a data dependence exists between two statements, when actually none exists.
Data Dependence: Definition

A data dependence, $S_1 \rightarrow S_2$, exists between CFG nodes $S_1$ and $S_2$ with respect to variable $X$ if and only if

1. there exists a path $P: S_1 \rightarrow S_2$ in CFG, with no intervening write to $X$, and

2. at least one of the following is true:

   (a) (flow) $X$ is written by $S_1$ and later read by $S_2$, or
   (b) (anti) $X$ is read by $S_1$ and later is written by $S_2$ or
   (c) (output) $X$ is written by $S_1$ and later written by $S_2$
Def/Use chaining for Data Dependence Analysis

A *def-use chain* links a definition $D$ (i.e. a write access of variable $X$) to each use $U$ (i.e. a read access), such that there is a path from $D$ to $U$ in $CFG$ that does not redefine $X$.

Similarly, a *use-def chain* links a use $U$ to a definition $D$, and a *def-def chain* links a definition $D$ to a definition $D'$ (with no intervening write to $X$ in all cases).

Def-use, use-def, and def-def chains can be computed by data flow analysis, and provide a simple but conservative way of enumerating flow, anti, and output data dependences.
Static single assignment (SSA) form

- Static single assignment (SSA) form provides a more efficient data structure for enumerating def-use, def-use and def-def chains.

- SSA form requires that each use be reached by a single def (when representing def-use information; analogous requirements are enforced for representing use-def and def-def information). Each def is treated as a new “name” for the variable.

- Each variable is assumed to have a dummy definition at the START node of the CFG.

- A $\phi$ function is used to capture the merge of multiple reaching definitions.
Dealing with Merge Points

If Cond
Then X <- 4
Else X <- 6

Use variable X several times

- Tricky situation since both defs can reach all subsequent uses; exact reaching def depends on whether Cond evaluated to true or not
- Keeping track of true and false cases separately is complicated and intractable (in the presence of nested conditionals)
The SSA approach

If Cond
Then $X_1 \leftarrow 4$
Else $X_2 \leftarrow 6$

$X_3 \leftarrow \phi(X_1, X_2)$

Use variable $X_3$ several times

• The SSA solution is to add a special $\emptyset$ function at each merge point

• The new $\emptyset$-def $X_3$ captures the merge of $X_1$ and $X_2$