CS738: Advanced Compiler Optimizations

Data Flow Analysis

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs738
Department of CSE, IIT Kanpur

Agenda

- Static analysis and compile-time optimizations
- For the next few lectures
- *Intraprocedural* Data Flow Analysis
  - Classical Examples
  - Components

Assumptions

- Intraprocedural: Restricted to a single function
- Input in 3-address format
- Unless otherwise specified

3-address Code Format

- Assignments

  x = y op z
  x = op y
  x = y
- Jump/control transfer

  goto L
  if x relop y goto L
- Statements can have label(s)

  L: . . .
- Arrays, Pointers and Functions to be added later when needed

Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
  - whether two identical expressions evaluate to same value
    - used in common subexpression elimination
  - whether the result of an assignment is used later
    - used by dead code elimination

Data Flow Abstraction

- Basic Blocks (BB)
  - sequence of 3-address code stmts
  - single entry at the first statement
  - single exit at the last statement
  - Typically we use “maximal” basic block (maximal sequence of such instructions)

Identifying Basic Blocks

- *Leader*: The first statement of a basic block
  - The first instruction of the program (procedure)
  - Target of a branch (conditional and unconditional goto)
  - Instruction immediately following a branch

Special Basic Blocks

- Two special BBs are added to simplify the analysis
  - empty (?) blocks!
- *Entry*: The first block to be executed for the procedure analyzed
- *Exit*: The last block to be executed

Data Flow Abstraction

- Control Flow Graph (CFG)
- A rooted directed graph $G = (N, E)$
- N = set of BBs
  - including *Entry*, *Exit*
- E = set of edges

CFG Edges

- Edge $B_1 \rightarrow B_2 \in E$ if control can transfer from B_1 to B_2
  - Fall through
  - Through jump (goto)
  - Edge from *Entry* to (all?) real first BB(s)
  - Edge to *Exit* from all last BBs
    - BBs containing return
    - Last real BB

Data Flow Abstraction: Control Flow Graph

- Graph representation of paths that program may exercise during execution
- Typically one graph per procedure
- Graphs for separate procedures have to be combined/connected for interprocedural analysis
  - Later!
  - Single procedure, single flow graph for now.

Data Flow Abstraction: Program Points

- Input state/Output state for Stmt
  - Program point before/after a stmt
  - Denoted IN[s] and OUT[s]
  - Within a basic block:
    - Program point after a stmt is same as the program point before the next stmt

Data Flow Abstraction: Program Points

- Input state/Output state for BBs
  - Program point before/after a bb
  - Denoted IN[B] and OUT[B]
  - For B_1 and B_2:
    - if there is an edge from B_1 to B_2 in CFG, then the program point *after* the last stmt of B_1 *may be* followed immediately by the program point *before* the first stmt of B_2.

Data Flow Abstraction: Execution Paths

- An execution path is of the form

$$p_1, p_2, p_3, \ldots, p_n$$

  where $p_i \rightarrow p_{i+1}$ are adjacent program points in the CFG.
- Infinite number of possible execution paths in practical programs.
- Paths having no finite upper bound on the length.
- Need to *summarize* the information at a program point with a finite set of facts.

Data Flow Schema

- Data flow values associated with each program point
  - Summarize all possible states at that point
- *Domain:* set of all possible data flow values
- Different domains for different analyses/optimizations

Data Flow Problem

- Constraints on data flow values
  - Transfer constraints
  - Control flow constraints
- **Aim:** To find a solution to the constraints
  - Multiple solutions possible
  - Trivial solutions, . . . , Exact solutions
- We typically compute approximate solution
  - Close to the exact solution (as close as possible!)
  - Why not exact solution?

Data Flow Constraints: Transfer Constraints

- Transfer functions
  - relationship between the data flow values before and after a stmt
- forward functions: Compute facts *after* a statement s from the facts available *before* s.
  - General form:

$$\text{OUT}[s] = f_s(\text{IN}[s])$$

- backward functions: Compute facts *before* a statement s from the facts available *after* s.
  - General form:

$$\text{IN}[s] = f_s(\text{OUT}[s])$$

- f_s depends on the statement and the analysis

Data Flow Constraints: Control Flow Constraints

- Relationship between the data flow values of two points that are related by program execution semantics
- For a basic block having n statements:

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], i = 1, 2, \ldots, n-1$$

- $\text{IN}[s_1]$, $\text{OUT}[s_n]$ to come later

Data Flow Constraints: Notations

- PRED (B): Set of predecessor BBs of block B in CFG
- SUCC (B): Set of successor BBs of block B in CFG
- $f \circ g$: Composition of functions f and g
- $\bigoplus$: An abstract operator denoting some way of combining facts present in a set .

Data Flow Constraints: Basic Blocks

- **Forward**
  - For B consisting of $s_1, s_2, \ldots, s_n$

$$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

  - Control flow constraints

$$\text{IN}[B] = \bigoplus_{P \in \text{PRED}(B)} \text{OUT}[P]$$

- **Backward**

$$f_B = f_{s_1} \circ f_{s_2} \circ \ldots \circ f_{s_n}$$

$$\text{IN}[B] = f_B(OUT[B])$$

$$\text{OUT}[B] = \bigoplus_{S \in \text{SUCC}(B)} IN[S]$$

Data Flow Equations

- Typical Equation

$$\text{OUT}[s] = \text{IN}[s] - \mathit{kill}[s] \cup \mathit{gen}[s]$$

  $\mathit{gen}(s)$: information generated

  $\mathit{kill}(s)$: information killed
- Example:

```
a = b*c // generates expression b * c
c = 5   // kills expression b*c
d = b*c // is b*c redundant here?
```

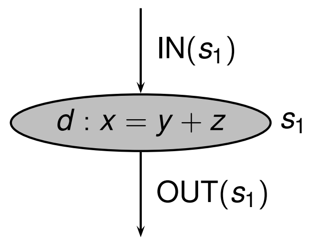
Example Data Flow Analysis

- Reaching Definitions Analysis
- Definition of a variable x: $x = \ldots$something$\ldots$
- Could be more complex (e.g. through pointers, references, implicit)

Reaching Definitions Analysis

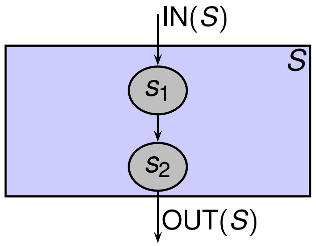
- A definition d reaches a point p if
  - there is a path from the point *immediately following* d to p
  - d is not “killed” along that path
  - “Kill” means redefinition of the left hand side (x in the earlier example)

RD Analysis of a Structured Program



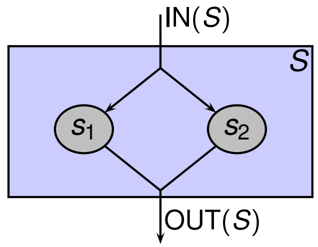
$$
\begin{aligned}
\text{OUT}(s_1) &= \text{IN}(s_1) - \text{KILL}(s_1) \cup \text{GEN}(s_1) \\
\text{GEN}(s_1) &= \{d\} \\
\text{KILL}(s_1) &= D_x - \{d\}, \text{where } D_x\text{: set of all definitions of } x \\
\text{KILL}(s_1) &= D_x\text{? will also work here} \\
&\quad\ \text{but may not work in general}
\end{aligned}
$$

RD Analysis of a Structured Program



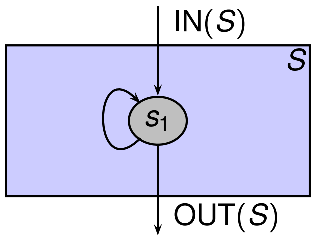
$$
\begin{aligned}
\mathrm{GEN}(S) &= \mathrm{GEN}(s_1) - \mathrm{KILL}(s_2) \cup \mathrm{GEN}(s_2)\\
\mathrm{KILL}(S) &= \mathrm{KILL}(s_1) - \mathrm{GEN}(s_2) \cup \mathrm{KILL}(s_2)\\
\mathrm{IN}(s_1) &= \mathrm{IN}(S)\\
\mathrm{IN}(s_2) &= \mathrm{OUT}(s_1)\\
\mathrm{OUT}(S) &= \mathrm{OUT}(s_2)
\end{aligned}
$$

RD Analysis of a Structured Program



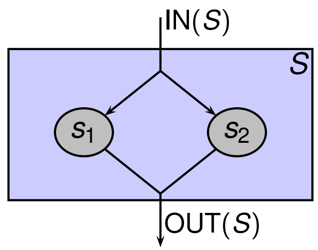
$$
\begin{aligned}
\mathrm{GEN}(S) &= \mathrm{GEN}(s_1) \cup \mathrm{GEN}(s_2)\\
\mathrm{KILL}(S) &= \mathrm{KILL}(s_1) \cap \mathrm{KILL}(s_2)\\
\mathrm{IN}(s_1) &= \mathrm{IN}(s_2) \quad = \quad \mathrm{IN}(S)\\
\mathrm{OUT}(S) &= \mathrm{OUT}(s_1) \cup \mathrm{OUT}(s_2)
\end{aligned}
$$

RD Analysis of a Structured Program



$$
\begin{aligned}
\mathrm{GEN}(S) &= \mathrm{GEN}(s_1)\\
\mathrm{KILL}(S) &= \mathrm{KILL}(s_1)\\
\mathrm{OUT}(S) &= \mathrm{OUT}(s_1)\\
\mathrm{IN}(s_1) &= \mathrm{IN}(S) \cup \mathrm{GEN}(s_1)
\end{aligned}
$$

RD Analysis is Approximate

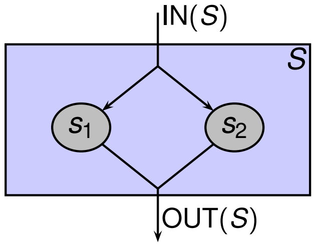


- Assumption: All paths are feasible.
- Example:

```
if (true) s1;
else      s2;
```

Fact		Computed		Actual
$\mathrm{GEN}(S)$	$=$	$\mathrm{GEN}(s_1) \cup \mathrm{GEN}(s_2)$	$\supseteq$	$\mathrm{GEN}(s_1)$
$\mathrm{KILL}(S)$	$=$	$\mathrm{KILL}(s_1) \cap \mathrm{KILL}(s_2)$	$\subseteq$	$\mathrm{KILL}(s_1)$

RD Analysis is Approximate



- Thus,

  true GEN(S) $\subseteq$ analysis GEN(S)
  true KILL(S) $\supseteq$ analysis KILL(S)
- More definitions computed to be reaching than actually do!
- Later we shall see that this is SAFE approximation
  - prevents optimizations
  - but NO wrong optimization

RD at BB level

- A definition d can reach the start of a block from any of its predecessor
  - if it reaches the end of some predecessor

$$\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$$

- A definition d reaches the end of a block if
  - either it is generated in the block
  - or it reaches block and not killed

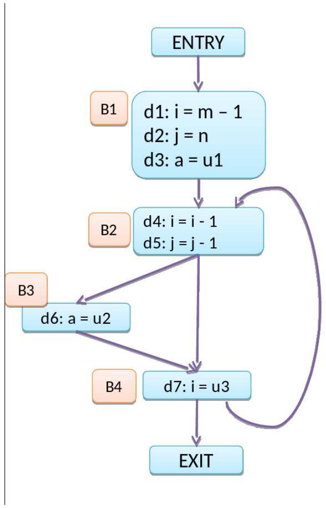
$$\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$$

Solving RD Constraints

- KILL & GEN known for each BB.
- A program with N BBs has $2N$ equations with $2N$ unknowns.
  - Solution is possible.
  - Iterative approach (on the next slide).

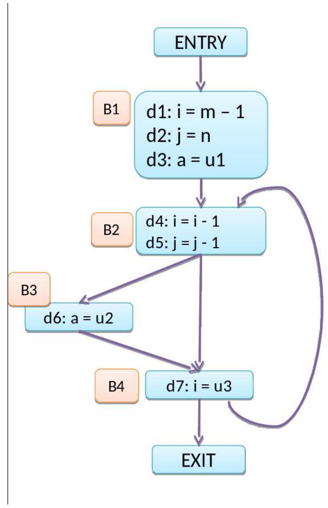
```
for each block B {
    OUT(B) = ∅;
}
OUT(Entry) = ∅; // note this for later discussion
change = true;
while (change) {
    change = false;
    for each block B other than Entry {
        IN(B) = ⋃_{P∈PRED(B)} OUT(P);
        oldOut = OUT(B);
        OUT(B) = IN(B) − KILL(B) ∪ GEN(B);
        if (OUT(B) ≠oldOut) then {
            change = true;
        }
    }
}
```

Reaching Definitions: Example



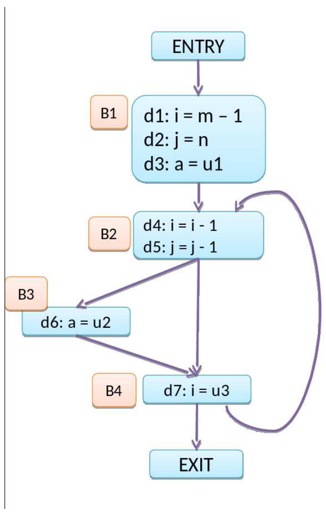
BB	GEN	KILL
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3	{d6}	{d3}
B4	{d7}	{d1, d4}

Reaching Definitions: Example



Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	∅	∅	∅	∅
1	IN	∅	d1, d2, d3	d3, d4, d5	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5	d4, d5, d6	d3, d5, d6, d7
2	IN	∅	d1, d2, d3, d5, d6, d7	d3, d4, d5, d6	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5, d6	d4, d5, d6	d3, d5, d6, d7
3	IN	∅	d1, d2, d3, d5, d6, d7	d3, d4, d5, d6	d3, d4, d5, d6
	OUT	d1, d2, d3	d3, d4, d5, d6	d4, d5, d6	d3, d5, d6, d7

Reaching Definitions: Bitvectors



a bit for each definition:

d1	d2	d3	d4	d5	d6	d7

Pass#	Pt	B1	B2	B3	B4
Init	IN	-	-	-	-
	OUT	0000000	0000000	0000000	0000000
1	IN	0000000	1110000	0011100	0011110
	OUT	1110000	0011100	0001110	0010111
2	IN	0000000	1110111	0011110	0011110
	OUT	1110000	0011110	0001110	0010111
3	IN	0000000	1110111	0011110	0011110
	OUT	1110000	0011110	0001110	0010111

Reaching Definitions: Bitvectors

- Set-theoretic definitions:

$$\mathrm{IN}(B) = \bigcup_{P \in \mathrm{PRED}(B)} \mathrm{OUT}(P)$$

$$\mathrm{OUT}(B) = \mathrm{IN}(B) - \mathrm{KILL}(B) \cup \mathrm{GEN}(B)$$

- Bitvector definitions:

$$\mathrm{IN}(B) = \bigvee_{P \in \mathrm{PRED}(B)} \mathrm{OUT}(P)$$

$$\mathrm{OUT}(B) = \mathrm{IN}(B) \wedge \neg\mathrm{KILL}(B) \vee \mathrm{GEN}(B)$$

- Bitwise $\vee, \wedge, \neg$ operators

Reaching Definitions: Application

Constant Folding

```
while changes occur {
   forall the stmts S of the program {
      foreach operand B of S {
         if there is a unique definition of B
         that reaches S and is a constant C {
            replace B by C in S;
            if all operands of S are constant {
               replace rhs by eval(rhs);
               mark definition as constant;
}}}}}
```

Reaching Definitions: Application

- Recall the approximation in reaching definition analysis
  true GEN(S) $\subseteq$ analysis GEN(S)
  true KILL(S) $\supseteq$ analysis KILL(S)
- Can it cause the application to infer
  - an expression as a constant when it is has different values for different executions?
  - an expression as not a constant when it is a constant for all executions?
- Safety? Profitability?

Reaching Definitions: Summary

- $\text{GEN}(B) = \left\{ d_x \;\middle|\; \begin{array}{l} d_x \text{ in } B \text{ defines variable } x \text{ and is not} \\ \text{followed by another definition of } x \text{ in } B \end{array} \right\}$
- $\text{KILL}(B) = \{d_x \mid B \text{ contains some definition of } x\}$
- $\text{IN}(B) = \bigcup_{P \in \text{PRED}(B)} \text{OUT}(P)$
- $\text{OUT}(B) = \text{IN}(B) - \text{KILL}(B) \cup \text{GEN}(B)$
- meet ($\bigwedge$) operator: The operator to combine information coming along different predecessors is $\cup$
- What about the *Entry* block?

Reaching Definitions: Summary

- Entry block has to be initialized specially:

$$\begin{aligned} \text{OUT}(\mathit{Entry}) &= \text{EntryInfo} \\ \text{EntryInfo} &= \emptyset \end{aligned}$$

- A better entry info could be:

$$\text{EntryInfo} = \{x = \mathit{undefined} \mid x \; \mathit{is\ a\ variable}\}$$

- Why?