Code analysis and transformation

DFA

Simone Campanoni
simonec@eecs.northwestern.edu
The spirit of my lectures
a.k.a. my teaching philosophy

• I’ll describe problems/opportunities
• I’ll describe concepts required to solve these problems
  (take advantage of these opportunities)
• I’ll describe the solutions that are based on these concepts

• I’ll describe new problems/opportunities
  • You’ll apply concepts/solutions learned in these lectures
    to solve the new problems/opportunities
    • Required to pass the homework
Data Flow Analysis outline

• 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? Very often the solution relies on designing an ad-hoc DFA
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

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

• Find an instruction $i$ that defines a variable with a constant expression

  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

CFA
DFA
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 0;}
    return x;
}
```

We need to know the “data-flow” of the program
Understand the data-flow requires understanding the control-flow
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)
Data Flow Analysis outline

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
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.

```
If (b > N)
b = b + 1
... = b
b = 1
b = 2
```

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”? 

```
... = b
```

```
b = 1
```
```
b = 2
```
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 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
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

- To do so, it relies on information specific to $i$
  - Encoded in GEN[$i$], KILL[$i$]
  - $fs$ uses GEN[$i$] and KILL[$i$] to compute its output

- GEN[$i$] and KILL[$i$] are DFA-specific and data/control flow independent!
DFA steps

1) Define the DFA-specific sets GEN[i] and KILL[i], for all i

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

3) Compute all IN[i] and OUT[i] following a DFA-generic algorithm
   \[
   \text{OUT}[i] = fs( \text{IN}[i] ) \\
   \text{IN}[i] = fs( \text{OUT}[i] )
   \]

Compilers have a data flow framework to help developing new DFAs
Data Flow Analysis outline

• Why do we need DFA? (opportunities)

• Introduction to DFA (concepts)

• A DFA example: reaching definitions (concept application)

• Implementation of DFA (actual implementation)
Optimization example: constant propagation

```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;
}
```

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

$\text{IN[return y]} = \{y=0\}$

$\text{IN[return x]} = \{x=0, \ x = x + N\}$
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 (invalidated) 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.

\[
\begin{align*}
D: & \quad v = 0 \\
J: & \quad v = v + n \\
X: & \quad \ldots = v
\end{align*}
\]

- GEN[$D$] = \{D\}
- KILL[$J$] = \{D\}
Computing INs and OUTs

GEN[0] = {}  
GEN[1] = {1}  
GEN[2] = {2}  
GEN[3] = {}  

IN = {}  
OUT = {x=0}  

- 0: int x,y  
  1: x = 0  
  2: y = 0  
  3: If (a > b)

- Forward or backward?
- $fs[i] = fs(IN[i])$
- $GEN[i] = what_i_generates$
- $KILL[i] = what_i_invalidates$
- $fs$ within a basic block?
- Let $i$ be an instruction and $p$ be its only predecessor
  $IN[i] = OUT[p]$
  $OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$

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 $IN[i]$ and $OUT[i]$ for all $i$ in that graph:

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

  $IN[\text{entry}] = \emptyset$

```
0: int x,y
1: x = 0
2: y = 0
3: if (a > b)
4: x = x+N
5: if (b > N)
```
Data Flow Analysis outline

• 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?
Iterative algorithm for reaching definitions

• Given $\text{GEN}[i]$, $\text{KILL}[i]$ for all instructions $i$, we compute $\text{IN}[i]$ and $\text{OUT}[i]$ for all $i$

\begin{align*}
\text{OUT[ENTRY]} &= \{ \} \\
\text{for (each instruction } i \text{ other than ENTRY) } &\text{ OUT}[i] = \{ \} \\
\text{while (changes to any OUT occur) } &\text{ for (each instruction } i \text{ other than ENTRY) } \\
&\text{ IN}[i] = \bigcup \text{ a predecessor of } i \text{ OUT}[p] \\
&\text{ OUT}[i] = \text{GEN}[i] \cup (\text{IN}[i] – \text{KILL}[i])
\end{align*}
### Reaching definition in action

| GEN\[0\] = {} | KILL\[0\] = {} |
| GEN\[1\] = \{1\} | KILL\[1\] = \{4\} |
| GEN\[2\] = \{2\} | KILL\[2\] = {} |
| GEN\[3\] = {} | KILL\[3\] = {} |
| GEN\[4\] = \{4\} | KILL\[4\] = \{1\} |
| GEN\[5\] = {} | KILL\[5\] = {} |

#### Why do we need to reach a fixed point?

| IN\[0\] = {} | OUT\[0\] = {} |
| IN\[1\] = {} | OUT\[1\] = 1 |
| IN\[2\] = \{1\} | OUT\[2\] = \{1,2\} |
| IN\[3\] = \{1,2\} | OUT\[3\] = \{1,2\} |
| IN\[4\] = \{1,2\} | OUT\[4\] = \{2,4\} |
| IN\[5\] = \{1,2,4\} | OUT\[5\] = \{1,2,4\} |

#### Done?

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

• Memory representation of data flow values 
• Operations performed on them 
• What is an element in a set?
Now that you know reaching definition

• It’s time for the homework H2
Can we optimize the analysis?

\[
\text{OUT[ENTRY]} = \{ \};
\]

for (each instruction \(i\) other than ENTRY) \(\text{OUT}[i] = \{ \};\)

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

Is this always necessary?
Optimization 1: basic blocks

\[ \text{OUT[ENTRY]} = \{ \}; \]
\[ \text{for (each basic block } B \text{ other than ENTRY) } \text{OUT}[B] = \{ \}; \]
\[ \text{while (changes to any OUT occur)} \]
\[ \text{for (each basic block } B \text{ other than ENTRY) } \]
\[ \text{IN}[B] = \bigcup p \text{ a predecessor of } B \ \text{OUT}[p]; \]
\[ \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]); \]
\[ \}
\]
\[ \}

\text{Contains all definitions in block } B \text{ that are visible immediately after } B

\text{Contains all definitions killed by instructions in block } B
Optimization 2: bit-sets

\[
\text{OUT}[\text{ENTRY}] = \{ \}; \\
\text{for (each basic block } B \text{ other than } \text{ENTRY}) \quad \text{OUT}[B] = \{ \}; \\
\text{while (changes to any OUT occur)} \\
\quad \text{for (each basic block } B \text{ other than } \text{ENTRY}) \{ \\
\quad \quad \text{IN}[B] = \bigcup p \text{ a predecessor of } B \text{ OUT}[p]; \\
\quad \quad \text{OUT}[B] = \text{GEN}[B] \bigcup (\text{IN}[B] - \text{KILL}[B]); \\
\quad \}
\]

\}
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

\[ \text{OUT}[\text{ENTRY}] = \{ \}; \]

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

while (changes to any \text{OUT} occur)

for (each basic block B other than ENTRY) {

\[ \text{IN}[B] = \bigcup p \text{ a predecessor of } B \text{ OUT}[p]; \]

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

}

}
Optimization 3: work list

\[
\text{OUT[ENTRY]} = \{ \}; \\
\text{for (each basic block } B \text{ other than ENTRY) } \text{OUT}[B] = \{ \}; \\
\text{workList} = \text{all basic blocks} \\
\text{while (workList isn't empty)} \\
\hspace{1em} B = \text{pick and remove a block from workList} \\
\hspace{1em} \text{oldOUT} = \text{OUT}[B] \\
\hspace{1em} \text{IN}[B] = \cup \text{a predecessor of } B \text{ OUT}[p]; \\
\hspace{1em} \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B]); \\
\hspace{1em} \text{if (oldOut != OUT}[B]) \text{workList} = \text{workList U \{all successors of B\}} \\
\]
Optimization 4: block order

\[
\text{OUT[ENTRY]} = \{ \}; \\
\text{for (each basic block B other than ENTRY)} \; \text{OUT}[B] = \{ \}; \\
\text{workList} = \text{all basic blocks} \\
\text{while (workList isn’t empty)} \\
\hspace{3em} B = \text{pick and remove a block from workList} \\
\hspace{3em} \text{oldOUT} = \text{OUT}[B] \\
\hspace{3em} \text{IN}[B] = \bigcup p \text{ a predecessor of } B \; \text{OUT}[p]; \\
\hspace{3em} \text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] – \text{KILL}[B]); \\
\hspace{3em} \text{if (oldOut } \neq \text{OUT}[B]) \text{workList} = \text{workList} \cup \{\text{all successors of B}\} \\
\]
• What we learned was for forward data-flow analysis

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

\[
\text{OUT}[\text{ENTRY}] = \{\};
\]

for (each instruction \(i\) other than ENTRY) \(\text{OUT}[i] = \{\};\)

while (changes to any \(\text{OUT}\) occur)

for (each instruction \(i\) other than ENTRY) {

\[
\text{IN}[i] = \cup p \text{ a predecessor of } i \text{ OUT}[p];
\]

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

}

}

• What about backward data-flow analysis?

\[ \text{IN}[s] = fs(\text{OUT}[s]) \]
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?