Lecture 33-34

Code Optimizations

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

April 25, 2025

Data Flow Analysis

- Global optimizations are based on dataflow analysis

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differs in the properties they compute

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differs in the properties they compute
- Example: constant propagation

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differs in the properties they compute
- Example: constant propagation
  - computes for each point in the program

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differs in the properties they compute
- Example: constant propagation
  - computes for each point in the program
  - for each variable used by the program

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differs in the properties they compute
- Example: constant propagation
  - computes for each point in the program
  - for each variable used by the program
  - whether variable has a unique constant value at that point

Data Flow Analysis

- Global optimizations are based on dataflow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differs in the properties they compute
- Example: constant propagation
  - computes for each point in the program
  - for each variable used by the program
  - whether variable has a unique constant value at that point
  - replace variable reference by a constant value

Data Flow Abstraction

- Find data flow value at each program point.

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point
- Forward and Backward Analysis

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point
- Forward and Backward Analysis
  - $OUT[s] = f_s(IN[s])$

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point
- Forward and Backward Analysis
  - $OUT[s] = f_s(IN[s])$
  - $IN[s] = f_s(OUT[s])$

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point
- Forward and Backward Analysis
  - $OUT[s] = f_s(IN[s])$
  - $IN[s] = f_s(OUT[s])$
- What if we can reach a program point from two locations?

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point
- Forward and Backward Analysis
  - $OUT[s] = f_s(IN[s])$
  - $IN[s] = f_s(OUT[s])$
- What if we can reach a program point from two locations?
  - $IN[B] = \cup_{P \text{ is predecessor of } B} OUT[P]$ in case of forward analysis

Data Flow Abstraction

- Find data flow value at each program point.
- IN[s] and OUT[s]
- Transfer function: defines the relation between data flow values before and after the program point
- Forward and Backward Analysis
  - $OUT[s] = f_s(IN[s])$
  - $IN[s] = f_s(OUT[s])$
- What if we can reach a program point from two locations?
  - $IN[B] = \cup_{P \text{ is predecessor of } B} OUT[P]$ in case of forward analysis
  - $OUT[B] = \cup_{S \text{ is successor of } B} IN[S]$ in case of backward analysis

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from d to p and `d is not` *killed along that path.*

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from `d` to `p` and `d is not` *killed along that path.*
- *Possible use case?*

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from `d` to `p` and `d is not` *killed along that path.*
- *Possible use case?*
  - *Variable used before assignment.*

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from `d` to `p` and `d is not` *killed along that path.*
- *Possible use case?*
  - *Variable used before assignment.*
  - *How to detect it?*

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from d to p and `d is not` *killed along that path.*
- *Possible use case?*
  - *Variable used before assignment.*
  - *How to detect it?*
  - *Assign a dummy definition to each variable; if any dummy definition is used, then the variable is used without a definition.*

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from `d` to `p` and `d is not` *killed along that path.*
- *Possible use case?*
  - *Variable used before assignment.*
  - *How to detect it?*
  - *Assign a dummy definition to each variable; if any dummy definition is used, then the variable is used without a definition.*
- $OUT[B] = gen_B \cup (IN[B] - kill_B)$

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from d to p and `d is not` *killed along that path.*
- *Possible use case?*
  - *Variable used before assignment.*
  - *How to detect it?*
  - *Assign a dummy definition to each variable; if any dummy definition is used, then the variable is used without a definition.*
- $OUT[B] = gen_B \cup (IN[B] - kill_B)$
- $IN[B] = \cup_{P \text{ is predecessor of } B} OUT[P]$

Reaching Definitions

- Where in the program each variable may have been defined when control reaches each point
- *d reaches a point p* if there is a path from d to p and `d is not` *killed along that path.*
- *Possible use case?*
  - *Variable used before assignment.*
  - *How to detect it?*
  - *Assign a dummy definition to each variable; if any dummy definition is used, then the variable is used without a definition.*
- $OUT[B] = gen_B \cup (IN[B] - kill_B)$
- $IN[B] = \cup_{P \text{ is predecessor of } B} OUT[P]$
- *Solve till the fix point is reached.*

Reaching definition is a forward analysis.

Live Variable Analysis

- Whether the value of `x could be used at program point p or not.`

Live Variable Analysis

- Whether the value of `x could be used at program point p or not.`
- def_B `set of variables defined in B prior to any use of that variable in B`

Live Variable Analysis

- Whether the value of `x could be used at program point p or not.`
- def_B `set of variables defined in B prior to any use of that variable in B`
- use_B `Value may be used in B prior to any definition of the variable`

Live Variable Analysis

- Whether the value of `x could be used at program point p or not.`
- def_B `set of variables defined in B prior to any use of that variable in B`
- use_B `Value may be used in B prior to any definition of the variable`
- $IN[EXIT] = \phi$

Live Variable Analysis

- Whether the value of `x could be used at program point p or not.`
- def_B `set of variables defined in B prior to any use of that variable in B`
- use_B `Value may be used in B prior to any definition of the variable`
- $IN[EXIT] = \phi$
- $IN_B = use_B \cup (OUT[B] - def_B)$

Live Variable Analysis

Live variable analysis is a backward analysis.

- Whether the value of `x could be used at program point p or not.`
- def_B `set of variables defined in B prior to any use of that variable in B`
- use_B `Value may be used in B prior to any definition of the variable`
- $IN[EXIT] = \phi$
- $IN_B = use_B \cup (OUT[B] - def_B)$
- $OUT[B] = \cup_{S \text{ is successor of } B} IN[S]$

Avaliable Expression

- An expression is *available at a point p if every path from the entry node to p evaluates that expression and its operands are not modified between the path.*

Avaliable Expression

- An expression is *available at a point p if every path from the entry node to p evaluates that expression and its operands are not modified between the path.*
- $OUT[ENTRY] = \phi$

Avaliable Expression

- An expression is *available at a point p if every path from the entry node to p evaluates that expression and its operands are not modified between the path.*
- $OUT[ENTRY] = \phi$
- $OUT[B] = gen_B \cup (IN[B] - kill_B)$

Avaliable Expression

Available expression is Forward analysis.

- An expression is *available at a point p if every path from the entry node to p evaluates that expression and its operands are not modified between the path.*
- $OUT[ENTRY] = \phi$
- $OUT[B] = gen_B \cup (IN[B] - kill_B)$
- $IN[B] = \cap_{P \text{ is predecessor of } B} OUT[P]$

Constant Propogation

if we found a variable has a constant value at a program point, we may replace variable with that value.

Data flow analysis can't be applied for constant propagation because transfer function is not distributive and it will not terminate the analysis.