Program Analysis

https://www.cse.iitb.ac.in/~karkare/cs618/

Code Optimizations

Amey Karkare
Dept of Computer Science and Engg
IIT Kanpur
Visiting IIT Bombay
karkare@cse.iitk.ac.in
karkare@cse.iitb.ac.in

Recap

Recap

- Optimizations
  - To improve efficiency of generated executable (time, space, resources …)
  - Maintain semantic equivalence

Recap

- Optimizations
  - To improve efficiency of generated executable (time, space, resources …)
  - Maintain semantic equivalence
- Two levels
  - Machine Independent
  - Machine Dependent

Machine Independent Optimizations

Machine Independent Optimizations

- Scope of optimizations

Machine Independent Optimizations

- Scope of optimizations
  - Local

Machine Independent Optimizations

- Scope of optimizations
  - Local
  - Global

Machine Independent Optimizations

- Scope of optimizations
  - Local
  - Global

  } Intraprocedural

Machine Independent Optimizations

- Scope of optimizations
  - Local
  - Global
  - Interprocedural

Intraprocedural

Local Optimizations

- Restricted to a basic block
- Simplifies the analysis
- Not all optimizations can be applied locally
  - E.g. Loop optimizations
- Gains are also limited
- Simplify global/interprocedural optimizations

Global Optimizations

Global Optimizations

- Typically restricted within a procedure/function
  - Could be restricted to a smaller scope, e.g. a loop

Global Optimizations

- Typically restricted within a procedure/function
  - Could be restricted to a smaller scope, e.g. a loop
- Most compiler implement up to global optimizations
  - Well founded theory
  - Practical gains

Interprocedural Optimizations

Interprocedural Optimizations

- Spans multiple procedures, files
  - In some cases multiple languages!

Interprocedural Optimizations

- Spans multiple procedures, files
  - In some cases multiple languages!
- Not as popular as global optimizations

Interprocedural Optimizations

- Spans multiple procedures, files
  - In some cases multiple languages!
- Not as popular as global optimizations
  - No single theory applicable to multiple scenarios
  - Time consuming

A Catalogue of Code Optimizations

Compile-time Evaluation

Compile-time Evaluation

- Move run-time actions to compile-time

Compile-time Evaluation

- Move run-time actions to compile-time
- Constant Folding:

Compile-time Evaluation

- Move run-time actions to compile-time
- Constant Folding:

```
Volume = 4/3*PI*r*r*r;
```

Compile-time Evaluation

- Move run-time actions to compile-time
- Constant Folding:

```
Volume = 4/3*PI*r*r*r;
```

  – Compute 4/3*PI at compile time

Compile-time Evaluation

- Move run-time actions to compile-time
- Constant Folding:

```
Volume = 4/3*PI*r*r*r;
```

  - Compute 4/3*PI at compile time
  - Applied very frequently for linearizing indices of multidimensional arrays

Compile-time Evaluation

- Move run-time actions to compile-time
- Constant Folding:

```
Volume = 4/3*PI*r*r*r;
```

  - Compute 4/3*PI at compile time
  - Applied very frequently for linearizing indices of multidimensional arrays
  - When can we apply it?

Compile-time Evaluation

Compile-time Evaluation

- Constant Propagation

Compile-time Evaluation

- Constant Propagation
  - Replace a variable by its “constant” value

Compile-time Evaluation

- Constant Propagation
  - Replace a variable by its “constant” value

```
i = 5;
…
j = i*4;
…
```

Replaced by

```
i = 5;
…
j = 5*4;
…
```

Compile-time Evaluation

- Constant Propagation
  - Replace a variable by its “constant” value

```
i = 5;
…
j = i*4;
…
```

Replaced by

```
i = 5;
…
j = 5*4;
…
```

  - May result in application of constant folding

Compile-time Evaluation

- Constant Propagation
  - Replace a variable by its “constant” value

```
i = 5;
…
j = i*4;
…
```

Replaced by

```
i = 5;
…
j = 5*4;
…
```

  - May result in application of constant folding
  - When can we apply it?

Common Subexpression Elimination

Common Subexpression Elimination

- Reuse a computation if already “available”

Common Subexpression Elimination

- Reuse a computation if already “available”

```
x = u+v;
…
y = u+v+w;
…
```

Replaced by

```
t0 = u+v;
x = t0;
…
y = t0+w;
…
```

Common Subexpression Elimination

- Reuse a computation if already "available"

```
x = u+v;
…
y = u+v+w;
…
```

Replaced by

```
t0 = u+v;
x = t0;
…
y = t0+w;
…
```

- When can we do it?

Copy Propagation

Copy Propagation

- Replace a variable by another
  - If they are guaranteed to have same value

Copy Propagation

- Replace a variable by another
  - If they are guaranteed to have same value

```
i = k;
…
j = i*4;
…
```

Replaced by

```
i = k;
…
j = k*4;
…
```

Copy Propagation

- Replace a variable by another
  - If they are guaranteed to have same value

```
i = k;
…
j = i*4;
…
```

Replaced by

```
i = k;
…
j = k*4;
…
```

  - May result in dead code, common subexpr, …

Copy Propagation

- Replace a variable by another
  - If they are guaranteed to have same value

```
i = k;
…
j = i*4;
…
```

Replaced by

```
i = k;
…
j = k*4;
…
```

  - May result in dead code, common subexpr, …
  - When can we apply it?

Code Movement

Code Movement

- Move the code in a program

Code Movement

- Move the code in a program
- Benefits:
  - Code size reduction
  - Reduction in the frequency of execution

Code Movement

- Move the code in a program
- Benefits:
  - Code size reduction
  - Reduction in the frequency of execution
- Allowed only if the meaning of the program does not change.

Code Movement

- Move the code in a program
- Benefits:
  - Code size reduction
  - Reduction in the frequency of execution
- Allowed only if the meaning of the program does not change.
  - May result in dead code, common subexpr, …

Code Movement

- Move the code in a program
- Benefits:
  - Code size reduction
  - Reduction in the frequency of execution
- Allowed only if the meaning of the program does not change.
  - May result in dead code, common subexpr, …
  - When can we apply it?

Code Movement

Code Movement

- Code size reduction

Code Movement

- Code size reduction

  Suppose op generates a large number of machine instructions

```
if (a < b)
  u = x op y;
else
  v = x op y;
```

Replaced by

```
t1 = x op y;
if (a < b)
  u = t1;
else
  v = t1;
```

Code Movement

Code Movement

- Execution frequency reduction

Code Movement

- Execution frequency reduction

```
if (a < b)
  u = …;
else
  v = x*y;
w = x*y;
```

Replaced by

```
if (a < b) {
 t2 = x*y;
 u  = …;
} else {
  t2 = x*y;
  v  = t2;
}
w = t2;
```

Code Movement

- Execution frequency reduction

```
if (a < b)
  u = …;
else
  v = x*y;
w = x*y;
```

Replaced by

```
if (a < b) {
 t2 = x*y;
 u  = …;
} else {
  t2 = x*y;
  v  = t2;
}
w = t2;
```

- When can we do it?

Loop Invariant Code Movement

Loop Invariant Code Movement

- Execution frequency reduction

Loop Invariant Code Movement

- Execution frequency reduction

```
for (…) {
    …
  u = a+b;
    …
}
```

Replaced by

```
t3 = a+b;
for (…) {
    …
  u  = t3;
    …
}
```

Loop Invariant Code Movement

- Execution frequency reduction

```
for (…) {
    …
  u = a+b;
    …
}
```

Replaced by

```
t3 = a+b;
for (…) {
    …
  u  = t3;
    …
}
```

- When can we do it?

Code Movement

Code Movement

- Safety of code motion

Code Movement

- Safety of code motion
- Profitability of code motion

Other optimizations

Other optimizations

- Dead code elimination
  - Remove unreachable, unused code.
  - Can we always do it?

Other optimizations

- Dead code elimination
  - Remove unreachable, unused code.
  - Can we always do it?
- Strength reduction
  - Use of *low strength* operators in place of *high strength* operators.
    - i*i instead of i^2, pow(i,2)
    - i<<1 instead of i*2
  - Typically performed for integers only (Why?)

Data Flow Analysis

Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths

Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:

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

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

Data Flow Abstraction

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

Data Flow Abstraction

Data Flow Abstraction

- Basic Blocks (bb)

Data Flow Abstraction

- Basic Blocks (bb)
- Input state/Output state for Stmt
  - Program point before/after a stmt
  - Denoted IN[s] and OUT[s]

Data Flow Abstraction

- Basic Blocks (bb)
- 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*