Fall 2017-2018 Compiler Principles
Lecture 8: Loop Optimizations

Copyright 2013 tagxedo.com

Roman Manevich
Ben-Gurion University of the Negev
Tentative syllabus

Front End
- Scanning
- Top-down Parsing (LL)
- Bottom-up Parsing (LR)

Intermediate Representation
- Operational Semantics
- Lowering

Optimizations
- Dataflow Analysis
- Loop Optimizations

Code Generation
- Register Allocation

mid-term

exam
• Dataflow analysis framework
  – Join semilattices
  – Chaotic iteration
  – Monotone transformers

• More optimizations
  – Reminder: Available expressions $\rightarrow$ common sub-expression elimination + copy propagation
  – Constant propagation $\rightarrow$ constant folding
AGENDA

• Loop optimizations
  – Reaching definitions → loop code motion + strength reduction via induction variables

• Dataflow summary
Loop optimizations

• Most of a program’s computations are done inside loops
  – Focus optimizations effort on loops
• The optimizations we’ve seen so far are independent of the control structure
• Some optimizations are specialized to loops
  – Loop-invariant code motion
  – Strength reduction via induction variables
• Require another type of analysis to find out where expressions get their values from
  – Reaching definitions
Loop invariant computation

```
y := t * 4
x < y + z
x := x + 1
```

start

```
y := ...
t := ...
z := ...
```

end
Loop invariant computation

start

\[ y := \ldots \]
\[ t := \ldots \]
\[ z := \ldots \]

\[ y := t \times 4 \]
\[ x < y + z \]

\[ x := x + 1 \]

end

t*4 and y+z have same value on each iteration
Code hoisting

start

\[ \begin{align*}
  y &:= \ldots \\
  t &:= \ldots \\
  z &:= \ldots \\
  y &:= t \times 4 \\
  w &:= y + z
\end{align*} \]

Only if \( y \) is not used before defined inside the loop

\[ x < w \]

\[ x := x + 1 \]

end
What reasoning did we use?

\[
y := t \times 4 \\
x < y + z \\
x := x + 1
\]

start

\[
\begin{align*}
y &:= \ldots \\
t &:= \ldots \\
z &:= \ldots
\end{align*}
\]

end

- Both \(t\) and \(z\) are defined only outside of loop.
- Constants are trivially loop-invariant.
- \(y\) is defined inside loop but it is loop invariant since \(t \times 4\) is loop-invariant.
What about now?

start

y := ...
t := ...
z := ...

Now t is not loop-invariant and so are t*4 and y

y := t * 4
x < y + z

x := x + 1
t := t + 1

date
Loop-invariant code motion

• \( d: t := a_1 \text{ op } a_2 \)

• \( a_1 \text{ op } a_2 \) loop-invariant (for a loop \( L \)) if computes the same value in each iteration
  – Hard to know in general

• Conservative approximation Each \( a_i \) is either:
  1. A constant, or
  2. All definitions of \( a_i \) that reach \( d \) are outside \( L \), or
  3. Only one definition of \( a_i \) reaches \( d \), and is loop-invariant itself

• Transformation: hoist the loop-invariant code outside of the loop
REACHING DEFINITIONS
ANALYSIS
Reaching definitions analysis

• **Definition:** An assignment \( d: t := R \) **reaches** a program location if there is a path from the definition to the program location, along which the defined variable is never redefined.

• Let’s define the corresponding dataflow analysis.
Reaching definitions analysis

• **Definition:** An assignment \( d: t := R \) reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined

• **Direction:** ?

• **Domain:** ?

• **Join operator:** ?

• **Transfer function:**
  \[ f_d: x := R(RD) = ? \]
  \[ f_d: not-a-def(RD) = ? \]

• **Initial value:** ?
Reaching definitions analysis

- **Definition:** An assignment $d: t := R$ reaches a program location if there is a path from the definition to the program location, along which the defined variable is never redefined.

- **Direction:** Forward

- **Domain:** sets of program locations that are definitions

- **Join operator:** union

- **Transfer function:**
  \[
  F[d: x := R](RD) = (RD \setminus \text{defs}(x, P)) \cup \{d\}
  \]
  \[
  F[d: \text{not-a-def}](RD) = RD
  \]
  - Where $\text{defs}(x, P)$ is the set of labels defining $x$ (statements of the form $x := R$) in $P$

- **Initial value:** $\{}$
Reaching definitions analysis

```
y := ...  
t := ...  
z := ...

y := t * 4  
x < y + z

x := x + 1

{ }

end
```
Reaching definitions analysis

```
d1: y := ...
d2: t := ...
d3: z := ...
d4: y := t * 4
   x < y + z
   x := x + 1
```

```java
{}
end
```
Initialization

- d1: y := ...
- d2: t :=...
- d3: z := ...
  - {}  
- d4: y := t * 4
  - x < y + z
  - {}  
- d5: x := x + 1
  - {}  
- {}
Iteration 1

start
{}

d1: y := ...
d2: t := ...
d3: z := ...
{
}
d4: y := t * 4
x < y + z
{}
{
}
d5: x := x + 1
{}
{
}
end
Iteration 1

```
start
{
}
```

```
\{d1\}
d1: y := ...
\{d1\}
```

```
\{d1, d2\}
d2: t := ...
\{d1, d2\}
```

```
\{d1, d2, d3\}
d3: z := ...
\{d1, d2, d3\}
```

```
\{d1\}
d4: y := t * 4
\{d1\}
```

```
\{d1, d2\}
x < y + z
\{d1, d2\}
```

```
\{d1, d2, d3\}
```

```
\{d1, d2, d3\}
d5: x := x + 1
\{d1, d2, d3\}
```

```
\{}\end
```
Iteration 2

start

{} 

\textbf{d1}: \ y := \ldots \ 
\quad \{d1\}

\textbf{d2}: \ t := \ldots \ 
\quad \{d1, d2\}

\textbf{d3}: \ z := \ldots \ 
\quad \{d1, d2, d3\}

\textbf{d4}: \ y := t \ast 4

\quad x < y + z 
\quad \{} 

\textbf{d5}: \ x := x + 1

\quad \{}

\textbf{end}
Iteration 2

start

{}  

\[d1: y := ... \]
\[\{d1\}\]
\n\[d2: t := ... \]
\[\{d1, d2\}\]
\n\[d3: z := ... \]
\[\{d1, d2, d3\}\]

\{d1, d2, d3\}
\n\[d4: y := t * 4 \]

\[x < y + z \]
\[\{\}\]

\[d5: x := x + 1 \]
\[\{\}\]

\{\}\n
end
Iteration 2

\[ y := t \times 4 \]
\[
\text{start} \\
\{\}
\]
\[
d1: y := \ldots \\
\{d1\}
\]
\[
d2: t := \ldots \\
\{d1, d2\}
\]
\[
d3: z := \ldots \\
\{d1, d2, d3\}
\]
\[
\{d1, d2, d3\}
\]
\[
d4: y := t \times 4 \\
\{d2, d3, d4\}
\]
\[
x < y + z \\
\{
\}
\]
\[
d5: x := x + 1 \\
\{
\}
\]
\[
\text{end} \\
\{\}
\]
Iteration 2

\begin{itemize}
\item \texttt{d1: y := \ldots }
\hspace{1cm} \{\texttt{d1}\}
\item \texttt{d2: t := \ldots }
\hspace{1cm} \{\texttt{d1, d2}\}
\item \texttt{d3: z := \ldots }
\hspace{1cm} \{\texttt{d1, d2, d3}\}
\item \texttt{d4: y := t * 4}
\hspace{1cm} \{\texttt{d2, d3, d4}\}
\hspace{1cm} x < y + z
\hspace{1cm} \{\texttt{d2, d3, d4}\}
\item \texttt{d5: x := x + 1}
\hspace{1cm} \{\}
\end{itemize}
Iteration 3

```
{}  
d1: y := ...  
   {d1}  
d2: t := ...  
   {d1, d2}  
d3: z := ...  
   {d1, d2, d3}

{d1, d2, d3}  
d4: y := t * 4  
   {d2, d3, d4}  
   x < y + z  
   {d2, d3, d4}

{d2, d3, d4}  
d5: x := x + 1  
   {}  
```

end
Iteration 3

start

{}
Iteration 4

start
{}

\{\}
d1: y := ...
  \{d1\}
d2: t := ...
  \{d1, d2\}
d3: z := ...
  \{d1, d2, d3\}

\{d1, d2, d3\}
d4: y := t * 4
\{d2, d3, d4\}
  x < y + z
\{d2, d3, d4\}

\{d2, d3, d4\}
d5: x := x + 1
\{d2, d3, d4, d5\}

\{\}
end
Iteration 4

```
start
{}  

{}  
  d1: y := ...  
  {d1}  
  d2: t := ...  
  {d1, d2}  
  d3: z := ...  
  {d1, d2, d3}  

{d1, d2, d3, d4, d5}  
  d4: y := t * 4  
  {d2, d3, d4}  
  x < y + z  
  {d2, d3, d4}  

{d2, d3, d4}  
  d5: x := x + 1  
  {d2, d3, d4, d5}  

{}  
end
```
Iteration 4

start

\{
\}
d1: y := ...
\{d1\}
d2: t := ...
\{d1, d2\}
d3: z := ...
\{d1, d2, d3\}

\{d1, d2, d3, d4, d5\}
d4: y := t * 4
\{d2, d3, d4, d5\}
x < y + z
\{d2, d3, d4, d5\}

\{d1, d2, d3, d4\}
d5: x := x + 1
\{d2, d3, d4, d5\}

\} end
Iteration 5

{d1, d2, d3, d4, d5}

\[ d4: y := t \times 4 \]
\[ x < y + z \]
\[ \{d2, d3, d4, d5\} \]

{d2, d3, d4, d5} end
Iteration 6

start

{}
Which expressions are loop invariant?

- y is defined only in d4 – inside of loop but depends on t and 4, both loop-invariant
- x is defined only in d5 – inside of loop so is not a loop-invariant
- t is defined only in d2 – outside of loop
- z is defined only in d3 – outside of loop
When is a transformation allowed

• Hoisting an assignment out of the loop body: as long as it doesn’t change the set of reaching definitions inside the loop body
• Adding an assignment to a temporary: Only if all the reaching definitions of the variables involved are outside the loop body
INFERRING LOOP INVARIANTS
Inferring loop-invariant expressions

• For a command \( c \) of the form \( t := a_1 \text{ op } a_2 \)
• A variable \( a_i \) is immediately loop-invariant if all reaching definitions \( \text{IN}[c] = \{d_1,...,d_k\} \) for \( a_i \) are outside of the loop
• \( \text{LOOP-INV} = \) immediately loop-invariant variables and constants
  \( \text{LOOP-INV} = \text{LOOP-INV} \cup \{x \mid d: x := a_1 \text{ op } a_2, d \text{ is in the loop, and both } a_1 \text{ and } a_2 \text{ are in LOOP-INV}\} \)
  — Iterate until fixed-point
• An expression is loop-invariant if all operands are loop-invariants
Computing LOOP-INV

start

{}
Computing LOOP-INV

\(\text{start} \{\} \quad \) (immediately) \(\text{LOOP-INV} = \{t\}\)

\(\{\} \quad d1: y := \ldots \quad \{d1\} \quad d2: t := \ldots \quad \{d1, d2\} \quad d3: z := \ldots \quad \{d1, d2, d3\} \)

\(\{d1, d2, d3, d4, d5\} \quad d4: y := t \times 4 \quad \{d2, d3, d4, d5\} \quad x < y + z \quad \{d2, d3, d4, d5\} \)

\(\{d2, d3, d4, d5\} \quad d5: x := x + 1 \quad \{d2, d3, d4, d5\} \)

\{d2, d3, d4\} \quad \text{end}
Computing LOOP-INV

start
{} 

(immediately)
LOOP-INV = \{t, z\}

\{d1, d2, d3, d4, d5\}
d4: y := t * 4
\{d2, d3, d4, d5\}
x < y + z
\{d2, d3, d4, d5\}

d5: x := x + 1
\{d2, d3, d4, d5\}

d2: t := ...
\{d1, d2\}
d3: z := ...
\{d1, d2, d3\}

d1: y := ...
\{d1\}

end
Computing LOOP-INV

```
d1: y := ...  
   {d1}

start  
   {}

... (immediately) LOOP-INV = \{t, z\} ...

{d1, d2, d3, d4, d5}  

x := y + z  
{d2, d3, d4, d5}  

{d2, d3, d4, d5}  

{d2, d3, d4, d5}  

d4: y := t * 4  
{d1, d2, d3, d4, d5}

{d2, d3, d4}  
end

```
Computing LOOP-INV

start

{} (immediately)
LOOP-INV = \{t, z\}

d1: y := ...
   \{d1\}
d2: t := ...
   \{d1, d2\}
d3: z := ...
   \{d1, d2, d3\}

\{d1, d2, d3, d4, d5\}
d4: y := t * 4
\{d2, d3, d4, d5\}
x < y + z
\{d2, d3, d4, d5\}

d5: x := x + 1
\{d2, d3, d4, d5\}

\{d2, d3, d4\}
end
Computing LOOP-INV

\[
\begin{align*}
\text{start} & \rightarrow \{\} \\
\{\} & \rightarrow \text{d1: } y := \ldots \\
\{d1\} & \rightarrow \text{d2: } t := \ldots \\
\{d1, d2\} & \rightarrow \text{d3: } z := \ldots \\
\{d1, d2, d3\} & \rightarrow \emptyset \\
\{d1, d2, d3, d4\} & \rightarrow \text{d4: } y := t \times 4 \\
\{d1, d2, d3, d4, d5\} & \rightarrow x < y + z \\
\{d2, d3, d4, d5\} & \rightarrow \text{x := x + 1} \\
\{d2, d3, d4, d5\} & \rightarrow \text{end}
\end{align*}
\]
Computing LOOP-INV

LOOP-INV = \{t, z, 4, y\}

Can move outside of loop only if every command in the loop sees only this definition (and not d1’s)
STRENGTH REDUCTION
VIA INDUCTION VARIABLES
Induction variables

while (i < x) {
  j := a + b * i
  a[j] := j
  i := i + 1
}

- `j` is a linear function of the induction variable with base `a` and multiplier `b`.
- `i` must receive definition from increment assignment.
- `i` is incremented by a loop-invariant expression on each iteration – this is called an induction variable.
Strength-reduction

\[ j := a + b \times (i-1) \]

while (i < x) {
    \[ j := j + b \]
    a[j] := j
    i := i + 1
}

Prepare initial value

Increment by multiplier
DATAFLOW SUMMARY
Summary of optimizations

- All of the analyses below use monotone transformers

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Enabled Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available Expressions</td>
<td>1. Common-subexpression elimination</td>
</tr>
<tr>
<td></td>
<td>2. Copy Propagation</td>
</tr>
<tr>
<td>Constant Propagation</td>
<td>Constant folding</td>
</tr>
<tr>
<td>Live Variables</td>
<td>Dead code elimination</td>
</tr>
<tr>
<td>Reaching Definitions (May + Must)</td>
<td>1. Loop-invariant code motion</td>
</tr>
<tr>
<td></td>
<td>2. Induction variable strength reduction</td>
</tr>
</tbody>
</table>
Analysis direction

• Forward
  – In the CFG, information is propagated from predecessors of a statement to its output
  – Properties depend on past computations
  – Examples: available expressions, constant propagating, reaching definitions

• Backward
  • In the CFG, information is propagated from successors of a statement to its output
  • Properties depend on future computations
  • Examples: Liveness analysis
Types of dataflow analysis

• May vs. Must
  – **Must analysis** – properties hold on **all paths** (join is set intersection)
    • Examples: available expressions, constant propagation
  – **May analysis** – properties hold on **some path** (join is set union)
    • Examples: liveness, reaching definitions
STRONG LIVENESS
The fixed-point of DCE

• DCE = dead code elimination
• Observation: DCE(P) does not always equal DCE(DCE(P))
  – DCE is not an idempotent transformation
Liveness example

{ b }
a := b;
{ a, b }
c := a;
{ a, b }
d := a + b;
{ a, b, d }
e := d;
{ a, b, e }
d := a;
{ b, d, e }
f := e;
{ b, d }
Foo(b, d)
{ }
Dead code elimination

```
{  b  }
{  a, b  }
{  a  }
{  a, b  }
{  a, b, d  }
{  a, b, e  }
{  a, b, e  }
{  b, d, e  }
{  b, d  }
{  } }
```

Can we find more dead assignments now?
Second liveness analysis

\{ b \}
a := b;

\{ a, b \}
d := a + b;
\{ a, b, d \}
e := d;
\{ a, b \}
d := a;

\{ b, d \}
Foo(b, d)
\{ \}
Second dead code elimination

\[
\begin{align*}
\{ \ b \ \} \\
a & := b; \\
\{ \ a, b \ \} \\
d & := a + b; \\
\{ \ a, b, d \ \} \\
e & := d; \\
\{ \ a, b \ \} \\
d & := a; \\
\{ \ b, d \ \} \\
\text{Foo}(b, d) \\
\{ \ \}
\end{align*}
\]

Can we find more dead assignments now?
Third liveness analysis

\{ b \}
a := b;

\{ a, b \}
d := a + b;

\{ a, b \}
d := a;

\{ b, d \}
Foo(b, d)
\{ \}
Third dead code elimination

\[
\{ b \} \\
a := b; \\
\{ a, b \} \\
d := a + b; \\
\{ a, b \} \\
d := a; \\
\{ b, d \} \\
\text{Foo}(b, d) \\
\{ \} \\
\{ \} 
\]
Fourth liveness analysis

\{ b \}

a := b;

\{ a, b \}
d := a;

\{ b, d \}

Foo(b, d)

\{ \}

No more dead statements
Loop example

```
{       }
IfZ x Goto L1
{       }

{       }
x := x - 1;
{       }
{       }
d := d + 1;
{       }
```

L1: Ret x
Strong liveness

• **Definition:** a variable $v$ is strongly live at label $L$ if ...

• What are the analysis parameters?

• SDCE = Strong Dead Code Elimination
  = DCE(StrongLiveness(P))
**Strong liveness transformer**

- **Reminder:** the liveness analysis transfer function for an assignment $x:=y+z$ is
  \[ F[x:=y+z](LV_{\text{out}}) = (LV_{\text{out}} \setminus \{x\}) \cup \{y, z\} \]

- **Definition:** a variable $v$ is **strongly live** at label $L$ if ...

- The strong liveness analysis transfer function for an assignment $x:=y+z$ is
  \[ F[x:=y+z](LV_{\text{out}}) = (LV_{\text{out}} \setminus \{x\}) \cup \{y, z \mid x \in LV_{\text{out}}\} \]
  – Is it monotone?
Strong liveness transformer

• **Reminder:** the liveness analysis transfer function for an assignment $x:=y+z$ is $F[x:=y+z](LV_{out}) = (LV_{out} \setminus \{x\}) \cup \{y, z\}$

• **Definition:** a variable $v$ is *strongly live* at label $L$ if
  1. it is an argument to a procedure, a return command, or a condition
  2. Used before assigned to a strongly live variable

• The strong liveness analysis transfer function for an assignment $x:=y+z$ is $F[x:=y+z](LV_{out}) = (LV_{out} \setminus \{x\}) \cup \{y, z \mid x \in LV_{out}\}$
  
  – Is it monotone?
Compute strong liveness

```
{ b }
a := b;
{ a, b }
c := a;
{ a, b }
d := a + b;
{ a, b }
e := d;
{ a, b }
d := a;
{ b, d }
f := e;
{ b, d }
Foo(b, d)
{ }
```
Eliminate dead code

```
{ b }
a := b;
{ a, b }
c := a;
{ a, b }
d := a + b;
{ a, b }
e := d;
{ a, b }
f := e;
{ b, d }
Foo(b, d)
{ } }
```
Eliminate dead code

{ b }
a := b;

{ a, b }
d := a;

{ b, d }
Foo(b, d)
{ }
Apply SDCE

Start

{ }
IfZ x Goto L1
{ }

{ }
x := x - 1;
{ }
d := d + 1;
{ }

{ }
L1: Ret x
{ }

End
COMPOSING ANALYSES
Accelerating copy propagation

• Can we compute $CP^*$ - the fixed point of applying copy propagation?
  – Yes: see question in the exam of 2016-B

• What about the fixed-point of $CF + CP + CSE + DCE$?
  – There is an algorithm by Sethi and Aho but it is limited to basic blocks
  – Is it extendible to CFGs? An open problem
Next lecture:

Register allocation