Loop Optimizations
April 6, 2011
CS 132: Compiler Design
LOOP OPTIMIZATIONS

✓ Loop Unrolling
✓ Hoisting loop-invariant computations
✓ Induction variable analysis
  ▶ Strength reduction
  ▶ Induction variable elimination
What is a (natural) loop?
Recall: Dominators

Recall:

✓ $d$ dominates $n$ if $d$ lies on every path from start to $n$.
✓ Non-start nodes have unique immediate dominators.
Recall: Dominators

Recall:

✓ *d* dominates *n* if *d* lies on every path from start to *n*.

✓ Non-start nodes have unique immediate dominators.

The Dominator
Geauga Lake, OH
**Computing Dominators**

\[
\text{dom}(\text{start}) = \{\text{start}\}
\]

\[
\text{dom}(n) = \{n\} \left\{ \begin{array}{c}
\cap \bigcup_{p \in \text{pred}(n)} \text{dom}(p) \\
\cap \bigcap_{p \in \text{pred}(n)} \text{dom}(p) \\
\cup \bigcup_{s \in \text{succ}(n)} \text{dom}(s) \\
\cap \bigcap_{s \in \text{succ}(n)} \text{dom}(s)
\end{array} \right\}
\]

\[
\cup \bigcup_{p \in \text{pred}(n)} \text{dom}(p) \\
\cup \bigcap_{p \in \text{pred}(n)} \text{dom}(p) \\
\cup \bigcup_{s \in \text{succ}(n)} \text{dom}(s) \\
\cap \bigcap_{s \in \text{succ}(n)} \text{dom}(s)
\]
Loops Formalized

A back edge is a directed edge where the target dominates the source.
Loops Formalized

A back edge is a directed edge where the target dominates the source.

The natural loop of a back edge \( t \rightarrow h \) is the set of nodes

- dominated by \( h \)
- that can reach \( t \) without going through \( h \).
Loops Formalized

A back edge is a directed edge where the target dominates the source.

The natural loop of a back edge \( t \rightarrow h \) is the set of nodes

- dominated by \( h \)
- that can reach \( t \) without going through \( h \).

A graph is reducible if all cycles are natural loops.

- i.e., removing back edges makes the graph acyclic.
- “Structured” code produces reducible graphs
- Reducible graphs permit faster dataflow analyses
Natural Loops? Reducible Graphs?
Can This Code Be Optimized?

c = getc();

for i = 1 to 20:

    m = i / (c+2);

    for j = i to 40:

        n = m * m;

        A[i,j] = (100 * i) * n * (c-2) * j;
Loop Invariant Computations

A computation is *loop-invariant* if every operand is:

✓ Constant

✓ Or, reached only by definitions outside the loop

✓ Or, reached by one loop-invariant definition.
Loop Invariant Computations

A computation is loop-invariant if every operand is:

✓ Constant
✓ Or, reached only by definitions outside the loop
✓ Or, reached by one loop-invariant definition.

How can we find reaching definitions?
IDENTIFY THE INVARIANT COMPUTATIONS

c = getc();

for i = 1 to 20:
    m = i / (c+2);

    for j = i to 40:
        n = m * m;

        A[i,j] = (100 * i) * n * (c-2) * j;
Hoisting

Invariant computations can (sometimes) be lifted out of a loop.

\[
\begin{align*}
t & \leftarrow 0 \\
\text{L1: } i & \leftarrow i + 1 \\
t & \leftarrow a + b \\
M[i] & \leftarrow t \\
\text{if } i < N & \text{ goto L1} \\
x & \leftarrow t \\
\end{align*}
\]

\[
\begin{align*}
t & \leftarrow 0 \\
\text{L1: } i & \leftarrow i + 1 \\
t & \leftarrow a + b \\
M[i] & \leftarrow t \\
\text{if } i < N & \text{ goto L1} \\
x & \leftarrow t \\
\end{align*}
\]

\[
\begin{align*}
t & \leftarrow 0 \\
\text{L1: } & \text{if } i \geq N \text{ goto L2} \\
i & \leftarrow i + 1 \\
t & \leftarrow a + b \\
M[i] & \leftarrow t \\
goto \text{ L1} \\
\text{L2: } x & \leftarrow t \\
\end{align*}
\]

\[
\begin{align*}
t & \leftarrow 0 \\
\text{L1: } M[j] & \leftarrow t \\
i & \leftarrow i + 1 \\
t & \leftarrow a + b \\
M[i] & \leftarrow t \\
\text{if } i < N & \text{ goto L1} \\
x & \leftarrow t \\
\end{align*}
\]
Hoisting A Definition

We can hoist \( t \leftarrow a + b \) if

✓ The definition dominates all loop exit targets where \( t \) is live,
Hoisting A Definition

We can hoist $t \leftarrow a + b$ if

✓ The definition dominates all loop exit targets where $t$ is live,
✓ and, there is only one definition of $t$ in the loop,
Hoisting A Definition

We can hoist $t \leftarrow a + b$ if

- The definition dominates all loop exit targets where $t$ is live,
- and, there is only on definition of $t$ in the loop,
- and, $t$ is not live entering the loop,
**Hoisting A Definition**

We can hoist $t \leftarrow a + b$ if

- The definition dominates all loop exit targets where $t$ is live,
- and, there is only one definition of $t$ in the loop,
- and, $t$ is not live entering the loop,
- and, there are not problems with side-effects
**Hoisting A Definition**

We can hoist \( t \leftarrow a \ + \ b \) if

- The definition dominates all loop exit targets where \( t \) is live,
- and, there is only one definition of \( t \) in the loop,
- and, \( t \) is not live entering the loop,
- and, there are no problems with side-effects
**Hoisting A Definition**

We can hoist \( t \leftarrow a + b \) if

- The definition dominates all loop exit targets where \( t \) is live,
- and, there is only one definition of \( t \) in the loop,
- and, \( t \) is not live entering the loop,
- and, there are not problems with side-effects

Sometimes it helps to treat

```java
while (e) s;
```

as

```java
if (e) {
    do
        s
    while (e);
}
```
Suppose we translate

```c
for (int i = 0; i < 10; ++i) sum += a[i];
```

as

```c
i ← 0
L1: j ← 4 * i
    k ← a + j
    x ← *a
    sum ← sum + x
    i ← i + 1
    if (i < 10) goto L1
// ...only sum is live afterwards...
```

How could you optimize this (without using left shift)?
Induction Variables

A variable $i$ is a basic induction variable in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.
Induction Variables

A variable $i$ is a basic induction variable in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A derived induction variable is a variable whose value is a linear (affine) function of an induction variable.
**Induction Variables**

A variable $i$ is a *basic induction variable* in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A *derived induction variable* is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy:

- There is one definition of $m$ in the loop
**Induction Variables**

A variable $i$ is a *basic induction variable* in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A *derived induction variable* is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy

 ✓ There is one definition of $m$ in the loop

 ✓ The definition is $m \leftarrow j \times c$ or $m \leftarrow j + c$ where $j$ is an induction variable and $c$ is loop invariant.
**Induction Variables**

A variable $i$ is a *basic induction variable* in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A *derived induction variable* is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy

- There is one definition of $m$ in the loop
- The definition is $m \leftarrow j \times c$ or $m \leftarrow j + c$ where $j$ is an induction variable and $c$ is loop invariant.
- And, if $j$ is derived from the basic induction $i$, then
INDUCTION VARIABLES

A variable $i$ is a basic induction variable in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A derived induction variable is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy

- There is one definition of $m$ in the loop
- The definition is $m \leftarrow j \times c$ or $m \leftarrow j + c$ where $j$ is an induction variable and $c$ is loop invariant.
- And, if $j$ is derived from the basic induction $i$, then
  - the only definition of $j$ reaching here is the one inside the loop
**Induction Variables**

A variable $i$ is a *basic induction variable* in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A *derived induction variable* is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy

- There is one definition of $m$ in the loop
- The definition is $m \leftarrow j \times c$ or $m \leftarrow j + c$ where $j$ is an induction variable and $c$ is loop invariant.
- And, if $j$ is derived from the basic induction $i$, then
  - the only definition of $j$ reaching here is the one inside the loop
  - There is no definition of $i$ on a path from the definition of $j$ to the definition of $m$. 
**Induction Variables**

A variable $i$ is a *basic induction variable* in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A *derived induction variable* is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy

- There is one definition of $m$ in the loop
- The definition is $m \leftarrow j \ast c$ or $m \leftarrow j + c$ where $j$ is an induction variable and $c$ is loop invariant.
- And, if $j$ is derived from the basic induction $i$, then
  - the only definition of $j$ reaching here is the one inside the loop
  - There is no definition of $i$ on a path from the definition of $j$ to the definition of $m$. 
**Induction Variables**

A variable $i$ is a *basic induction variable* in a loop if the only definitions of in the loop are of the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

A *derived induction variable* is a variable whose value is a linear (affine) function of an induction variable.

More formally, a variable $m$ must satisfy

- There is one definition of $m$ in the loop
- The definition is $m \leftarrow j \times c$ or $m \leftarrow j + c$ where $j$ is an induction variable and $c$ is loop invariant.
- And, if $j$ is derived from the basic induction $i$, then
  - the only definition of $j$ reaching here is the one inside the loop
  - There is no definition of $i$ on a path from the definition of $j$ to the definition of $m$.

We say that $m$ and $j$ are *in the same family*. 
**Strength Reduction**

*General term for replacing expensive operations with cheap ones.*
Strength Reduction

General term for replacing expensive operations with cheap ones.

In the context of induction variables, for each derived \( j = a + b \times i \):

✓ Create a new variable \( j' \) that closely tracks \( i \).
**Strength Reduction**

General term for replacing expensive operations with cheap ones.

In the context of induction variables, for each derived $j = a + b \times i$:

✓ Create a new variable $j'$ that closely tracks $i$.

✓ E.g., After each $i \leftarrow i + c$ in the loop add $j' \leftarrow j' + b \times c$
Strength Reduction

General term for replacing expensive operations with cheap ones.

In the context of induction variables, for each derived \( j = a + b \cdot i \):
- ✓ Create a new variable \( j' \) that closely tracks \( i \).
- ✓ E.g., After each \( i \leftarrow i + c \) in the loop add \( j' \leftarrow j' + b \cdot c \)
- ✓ Replace the definition of \( j \) with \( j \leftarrow j' \)
Exercise

Apply strength reduction to the previous for-loop.

```plaintext
i ← 0
L1: j ← 4 * i
k ← a + j
x ← *a
sum ← sum + x
i ← i + 1
if (i < 10) goto L1
// ...only sum is live afterwards...
```
Dead Code and Useless Variables

A variable is *dead* if it will never be used.

A variable in a loop is *useless* if it is dead at all loop exits, and is only used to define itself.

A variable in a loop is *almost useless* if

- it is used only in comparisons against loop-invariant values and in definitions of itself
- and, there is another induction variable in the same family that is not useless.

How do these concepts help?