Loop Optimizations

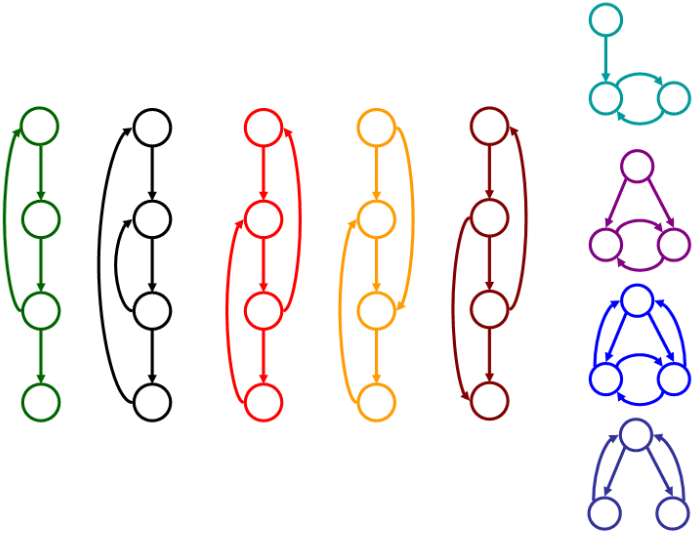
April 6, 2011

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:

- ✓ $\mathbf{d}$ dominates $\mathbf{n}$ if $\mathbf{d}$ lies on *every* path from *start* to $\mathbf{n}$.
- ✓ 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

$$
\begin{aligned}
\mathit{dom}(\mathit{start}) &= \{\mathit{start}\} \\
\mathit{dom}(n) &= \{n\} \left\{ \begin{array}{l}
\cap \bigcup_{p \in \mathit{pred}(n)} \mathit{dom}(p) \\
\cap \bigcap_{p \in \mathit{pred}(n)} \mathit{dom}(p) \\
\cap \bigcup_{s \in \mathit{succ}(n)} \mathit{dom}(s) \\
\cap \bigcap_{s \in \mathit{succ}(n)} \mathit{dom}(s) \\
\cup \bigcup_{p \in \mathit{pred}(n)} \mathit{dom}(p) \\
\cup \bigcap_{p \in \mathit{pred}(n)} \mathit{dom}(p) \\
\cup \bigcup_{s \in \mathit{succ}(n)} \mathit{dom}(s) \\
\cup \bigcap_{s \in \mathit{succ}(n)} \mathit{dom}(s)
\end{array} \right.
\end{aligned}
$$

Loops Formalized

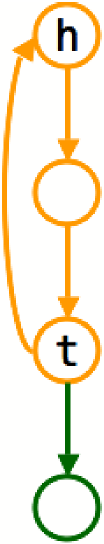
A *back edge* is a directed edge
where the target dominates the source.

Loops Formalized

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

The *natural loop* of a back edge $\mathrm{t} \rightarrow \mathrm{h}$ is the set of nodes

- ✓ dominated by h
- ✓ that can reach t without going through h.



Loops Formalized

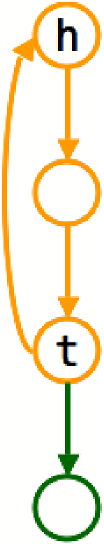
A *back edge* is a directed edge
where the target dominates the source.

The *natural loop* of a back edge $\mathrm{t} \rightarrow \mathrm{h}$ is the set of nodes

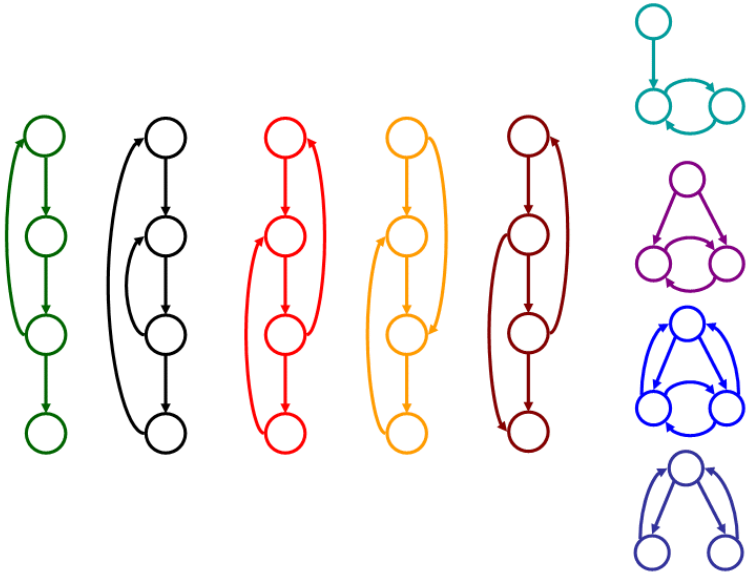
- ✓ dominated by h
- ✓ that can reach t without going through h.

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.

```
    t ← 0
L1: i ← i + 1
    t ← a + b
    M[i] ← t
    if i < N goto L1
    x ← t
```

```
    t ← 0
L1: i ← i + 1
    t ← a + b
    M[i] ← t
    t ← 0
    M[j] ← t
    if i < N goto L1
```

```
    t ← 0
L1: if i >= N goto L2
    i ← i + 1
    t ← a + b
    M[i] ← t
    goto L1
L2: x ← t
```

```
    t ← 0
L1: M[j] ← t
    i ← i + 1
    t ← a + b
    M[i] ← t
    if i < N goto L1
    x ← t
```

Hoisting A Definition

We can hoist `t ← a + b` if

- ✓ The definition dominates all loop exit targets where `t` is live,

Hoisting A Definition

We can hoist `t ← 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,

Hoisting A Definition

We can hoist `t ← 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 ← 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,
- ✓ and, there are not problems with side-effects

Hoisting A Definition

We can hoist `t ← 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,
- ✓ and, there are not problems with side-effects

Hoisting A Definition

We can hoist `t ← 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,
- ✓ and, there are not problems with side-effects

Sometimes it helps to treat

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

as

```
if (e) {
   do
       s
   while (e);
}
```

Another Example

Suppose we translate

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

as

```
    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 ← i + c` or `i ← 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 $\texttt{i} \leftarrow \texttt{i + c}$ or $\texttt{i} \leftarrow \texttt{i - c}$ 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 $\mathtt{i} \leftarrow \mathtt{i} + \mathtt{c}$ or $\mathtt{i} \leftarrow \mathtt{i} - \mathtt{c}$ 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 * 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 $m \leftarrow j * c$ 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 $\texttt{i} \leftarrow \texttt{i + c}$ or $\texttt{i} \leftarrow \texttt{i - c}$ 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 $\texttt{m} \leftarrow \texttt{j * c}$ or $\texttt{m} \leftarrow \texttt{j + c}$ 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 ← i + c` or `i ← 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 ← j * c` or `m ← 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 ← i + c` or `i ← 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 ← j * c` or `m ← 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 ← i + c` or `i ← 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 ← j * c` or `m ← 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 * 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 * 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 * c`

Strength Reduction

General term for replacing expensive operations with cheap ones.

In the context of induction variables, for each derived `j == a + b * i`:

- ✓ Create a new variable `j'` that closely tracks `i`.
- ✓ E.g., After each `i` ← `i + c` in the loop add `j'` ← `j' + b * c`
- ✓ Replace the definition of `j` with `j` ← `j'`

Exercise

Apply strength reduction to the previous for-loop.

```
    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?