Loops are everywhere

⇒ worthwhile target for optimization
Loops

**Definition: Loop**

A loop with header $h$ is a set $L$ of nodes in a CFG such that
- $h \in L$
- $(\forall s \in L)$ exists path from $h$ to $s$
- $(\forall s \in L)$ exists path from $s$ to $h$
- $(\forall t \notin L) (\forall s \in L)$ if there is an edge from $t$ to $s$, then $s = h$

**Special loop nodes**
- A loop entry node has a predecessor outside the loop.
- A loop exit node has a successor outside the loop.
Example Loops

1 -> 1
1 -> 2
2 -> 3
3 -> 2
3 -> 1
Example Loops

18-1a
Example Loops

18-1b
Example Loops
18-1d
Example Loops
int isPrime (int n) {
    i = 2;
    do {
        j = 2;
        do {
            if (i*j==n) {
                return 0;
            } else {
                j = j+1;
            }
        } while (j<n);
        i = i+1;
    } while (i<n);
    return 1;
}
Reducible Flow Graphs

- Arbitrary flow graphs: Spaghetti code
- Reducible flow graphs arise from structured control
  - if-then-else
  - while-do
  - repeat-until
  - for
  - break (multi-level)
Irreducible Flow Graphs

18-2a: Not a loop
Irreducible Flow Graphs

18-2b: Not a loop
Irreducible Flow Graphs

18-2c: Not a loop

- Reduces to 18-2a: collapse edges \((x, y)\) where \(x\) is the only predecessor of \(y\)
- A flow graph is irreducible if exhaustive collapsing leads to a subgraph like 18-2a.
Outline

1. Loops
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Dominators

Objective
Find all loops in flow graph

Assumption
Each CFG has unique entry node $s_0$ without predecessors

Domination relation
A node $d$ dominates a node $n$ if every path from $s_0$ to $n$ must go through $d$.

Remark
Domination is reflexive
Lemma
Let \( n \) be a node with predecessors \( p_1, \ldots, p_k \) and \( d \neq n \) a node. \( d \) dominates \( n \) iff \( (\forall 1 \leq i \leq k) \) \( d \) dominates \( p_i \).

Domination equation
Let \( D[n] \) be the set of nodes that dominate \( n \).

\[
D[n] = \{n\} \cup \bigcap_{p \in \text{pred}[n]} D[p]
\]

- Solve by fixed point iteration
- Start with \( (\forall n \in N) \ D[n] = N \) (all nodes in the CFG)
- Observe that \( D[s_0] = \{s_0\} \) because \( \text{pred}(s_0) = \emptyset \)
- Watch out for unreachable nodes
Immediate Dominators

Theorem
Let $G$ be a connected, rooted graph. If $d$ dominates $n$ and $e$ dominates $n$, then either $d$ dominates $e$ or $e$ dominates $d$.

- **Proof:** by contradiction
- **Consequence:** Each node $n \neq s_0$ has one immediate dominator $idom(n)$ such that
  1. $idom(n) \neq n$
  2. $idom(n)$ dominates $n$
  3. $idom(n)$ does not dominate another dominator of $n$
The **dominator tree** is a directed graph where the nodes are the nodes of the CFG and there is an edge $(x, y)$ if $x = idom(y)$.

- **back edge** in CFG: from $n$ to $h$ so that $h$ dominates $n$
Finding Loops

Natural Loop

The natural loop of a back edge \((n, h)\) where \(h\) dominates \(n\) is the set of nodes \(x\) such that

- \(h\) dominates \(x\)
- exists path from \(x\) to \(n\) not containing \(h\)

\(h\) is the header of this natural loop.
Nested Loops

Nested Loop
If $A$ and $B$ are loops with headers $a \neq b$ and $b \in A$, then $B \subseteq A$. Loop $B$ is nested within $A$. $B$ is the inner loop.

Algorithm: Loop-nest Tree

1. Compute the dominators of the CFG
2. Compute the dominator tree
3. Find all natural loops with their headers
4. For each loop header $h$ merge all natural loops of $h$ into a single loop $\text{loop}[h]$
5. Construct the tree of loop headers such that $h_1$ is above $h_2$ if $h_2 \in \text{loop}[h_1]$

- Leaves are innermost loops
- Procedure body is pseudo-loop at root of loop-nest tree
A Loop-Nest Tree

```
1
  6, 7, 11, 12

2
  3, 4

5
  10

8
  9
```
Adding a Loop Preheader

- Loop optimizations need a CFG node before the loop as a target to move code out of the loop
  ⇒ add preheader node like $P$ in example
Outline

1. Loops
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Suppose $t \leftarrow a \oplus b$ occurs in a loop.

If $a$ and $b$ have the same value for each iteration of the loop, then $t$ always gets the same value.

⇒ $t$’s definition is loop-invariant, but its computation is repeated on each iteration.

Goals

- Detect such loop-invariant definitions
- Hoist them out of the loop
Approximation to Loop-Invariance

Loop-Invariance

The definition \( d : t \leftarrow a_1 \oplus a_2 \) is loop-invariant for loop \( L \) if \( d \in L \) and, for each \( a_i \), one of the following conditions holds:

1. \( a_i \) is a constant,
2. all definitions of \( a_i \) that reach \( d \) are outside of \( L \), or
3. only one definition of \( a_i \) reaches \( d \) and that definition is loop-invariant.

Algorithm: Loop-Invariance

1. Identify all definitions whose operands are constant or defined outside the loop
2. Add loop-invariant definitions until a fixed point is reached
Suppose $t \leftarrow a \oplus b$ is loop-invariant.
Can we hoist it out of the loop?

<table>
<thead>
<tr>
<th>$L_0$</th>
<th>$L_0$</th>
<th>$L_0$</th>
<th>$L_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
</tr>
<tr>
<td>$L_1$</td>
<td>$L_1$</td>
<td>$L_1$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
<td>if $i \geq N$ goto $L_2$</td>
<td>$i \leftarrow i + 1$</td>
<td>$i \leftarrow i + 1$</td>
</tr>
<tr>
<td>$t \leftarrow a \oplus b$</td>
<td>$t \leftarrow a \oplus b$</td>
<td>$M[i] \leftarrow t$</td>
<td>$M[j] \leftarrow t$</td>
</tr>
<tr>
<td>$M[i] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
</tr>
<tr>
<td>if $i &lt; N$ goto $L_1$</td>
<td>goto $L_1$</td>
<td>$M[j] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
</tr>
</tbody>
</table>
- Suppose $t \leftarrow a \oplus b$ is loop-invariant.
- Can we hoist it out of the loop?

<table>
<thead>
<tr>
<th>$L_0$</th>
<th>$L_0$</th>
<th>$L_0$</th>
<th>$L_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
</tr>
<tr>
<td>$L_1$</td>
<td>$L_1$</td>
<td>$L_1$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
<td>if $i \geq N$ goto $L_2$</td>
<td>$i \leftarrow i + 1$</td>
<td>$i \leftarrow i + 1$</td>
</tr>
<tr>
<td>$t \leftarrow a \oplus b$</td>
<td>$t \leftarrow a \oplus b$</td>
<td>$M[i] \leftarrow t$</td>
<td>$t \leftarrow a \oplus b$</td>
</tr>
<tr>
<td>$M[i] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
</tr>
<tr>
<td>if $i &lt; N$ goto $L_1$</td>
<td>goto $L_1$</td>
<td>if $i &lt; N$ goto $L_1$</td>
<td>if $i &lt; N$ goto $L_1$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
</tr>
</tbody>
</table>

- $L_0$:
  - $t \leftarrow 0$
  - $L_1$:
    - $i \leftarrow i + 1$
    - $t \leftarrow a \oplus b$
    - $M[i] \leftarrow t$
    - if $i < N$ goto $L_1$
  - $L_2$:
    - $x \leftarrow t$

- $L_0$:
  - $t \leftarrow 0$
  - $L_1$:
    - if $i \geq N$ goto $L_2$
    - $i \leftarrow i + 1$
    - $t \leftarrow a \oplus b$
    - $M[i] \leftarrow t$
    - $M[i] \leftarrow t$
    - if $i < N$ goto $L_1$
  - $L_2$:
    - $x \leftarrow t$

- $L_0$:
  - $t \leftarrow 0$
  - $L_1$:
    - $M[j] \leftarrow t$
    - $i \leftarrow i + 1$
    - $t \leftarrow a \oplus b$
    - $M[i] \leftarrow t$
    - if $i < N$ goto $L_1$
  - $L_2$:
    - $x \leftarrow t$
Criteria for hoisting

A loop-invariant definition $d : t \leftarrow a \oplus b$ can be hoisted to the end of its loop’s preheader if all of the following hold:

1. $d$ dominates all loop exits at which $t$ is live-out
2. there is only one definition of $t$ in the loop
3. $t$ is not live-out at the loop preheader

Attention: arithmetic exceptions, side effects of $\oplus$

Condition 1 often prevents hoisting from while loops: transform into repeat-until loops.
Outline

1. Loops
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
C-code for summation of a long array

```c
long sum(long a[], int n) {
    long s = 0;
    int i = 0;
    while (i<n) {
        s += a[i];
        i ++;
    }
    return s;
}
```
Consider the corresponding IR

\[
\begin{align*}
    s & \leftarrow 0 \\
    i & \leftarrow 0 \\
    \text{\textbf{L}1 :} & \quad \text{if } i \geq n \text{ goto } L_2 \\
    j & \leftarrow i \cdot 4 \\
    k & \leftarrow j + a \\
    x & \leftarrow M[k] \\
    s & \leftarrow s + x \\
    i & \leftarrow i + 1 \\
    \text{goto } L_1 \\
    \text{\textbf{L}2}
\end{align*}
\]
Consider the corresponding IR

\[
\begin{align*}
&\quad s \leftarrow 0 \\
&\quad i \leftarrow 0 \\
L_1 : &\quad \text{if } i \geq n \text{ goto } L_2 \\
&\quad j \leftarrow i \cdot 4 \\
&\quad k \leftarrow j + a \\
&\quad x \leftarrow M[k] \\
&\quad s \leftarrow s + x \\
&\quad i \leftarrow i + 1 \\
&\quad \text{goto } L_1 \\
L_2 &
\end{align*}
\]

before

\[
\begin{align*}
&\quad s \leftarrow 0 \\
&\quad k' \leftarrow a \\
&\quad b \leftarrow n \cdot 4 \\
&\quad c \leftarrow a + b \\
L_1 : &\quad \text{if } k' \geq c \text{ goto } L_2 \\
&\quad x \leftarrow M[k'] \\
&\quad s \leftarrow s + x \\
&\quad k' \leftarrow k' + 4 \\
&\quad \text{goto } L_1 \\
L_2 &
\end{align*}
\]

after
Induction Variables

- **Induction-variable analysis:**
  identify induction variables and relations among them

- **Strength reduction:**
  replace expensive operation (e.g., multiplication) by cheap operation (e.g., addition)

- **Induction-variable elimination:**
  remove dependent induction variables
A basic induction variable is directly incremented
A derived induction variable is computed from other induction variables
Describe an induction variable \( b' \) by a triple \((b, o, f)\), where
- \( b \) is a basic induction variable
- \( o \) is an offset
- \( f \) is a factor

so that \( b' = o + f \cdot b \).

A linear induction variable changes by the same amount in every iteration.
Induction Variables in the Example

- $i$ is a basic induction variable described by $(i, 0, 1)$
- $j$ is a derived induction variable:
  
  after $j ← i \cdot 4$, it is described by $(i, 0, 4)$
- $k$ is a derived induction variable:
  
  after $k ← j + a$, it is described by $(i, a, 4)$
Non-linear Induction Variables

\begin{equation*}
\begin{aligned}
  & s \leftarrow 0 \\
  & L_1 : \quad \text{if } s > 0 \text{ goto } L_2 \\
  & \quad i \leftarrow i + b \\
  & \quad j \leftarrow i \cdot 4 \\
  & \quad x \leftarrow M[j] \\
  & \quad s \leftarrow s - x \\
  & \quad \text{goto } L_1 \\
  & L_2 : \quad i \leftarrow i + 1 \\
  & \quad s \leftarrow s + j \\
  & \quad \text{if } i < n \text{ goto } L_1
\end{aligned}
\end{equation*}
Non-linear Induction Variables

\begin{align*}
\text{s} & \leftarrow 0 \\
L_1 : & \text{ if } s > 0 \text{ goto } L_2 \\
& i \leftarrow i + b \\
& j \leftarrow i \cdot 4 \\
& x \leftarrow M[j] \\
& s \leftarrow s - x \\
& \text{goto } L_1 \\
L_2 : & i \leftarrow i + 1 \\
& s \leftarrow s + j \\
& \text{if } i < n \text{ goto } L_1
\end{align*}

\begin{align*}
\text{s} & \leftarrow 0 \\
\text{j'} & \leftarrow i \cdot 4 \\
\text{b'} & \leftarrow b \cdot 4 \\
\text{n'} & \leftarrow n \cdot 4 \\
L_1 : & \text{ if } s > 0 \text{ goto } L_2 \\
& j' \leftarrow j' + b' \\
& j \leftarrow j' \\
& x \leftarrow M[j] \\
& s \leftarrow s - x \\
& \text{goto } L_1 \\
L_2 : & j' \leftarrow j' + 4 \\
& s \leftarrow s + j \\
& \text{if } j' < n' \text{ goto } L_1
\end{align*}

before \hspace{5cm} after
Detection of Induction Variables

**Basic Induction Variable (in the family of $i$)**

Variable $i$ is a **basic induction variable** if all definitions of $i$ in loop $L$ have the form $i \leftarrow i \pm c$ where $c$ is loop-invariant.

**Derived Induction Variable**

Variable $k$ is a **derived ind. var.** in the family of $i$ in loop $L$ if

1. there is exactly one definition of $k$ in $L$ of the form $k \leftarrow j \cdot c$ or $k \leftarrow j + d$ where $j$ is an induction variable in the family of $i$ and $c, d$ are loop-invariant

2. if $j$ is a derived induction variable in the family of $i$, then
   - only the definition of $j$ in $L$ reaches (the definition of) $k$
   - there is no definition of $i$ on any path between the definition of $j$ and the definition of $k$

3. If $j$ is described by $(i, a, b)$, then $k$ is described by $(i, a \cdot c, b \cdot c)$ or $(i, a + d, b)$, respectively.
Strength Reduction

- Often multiplication is more expensive than addition
  ⇒ Replace the definition $j \leftarrow i \cdot c$ of a derived induction variable by an addition

Procedure

- For each derived induction variable $j \sim (i, a, b)$ create new variable $j'$
- After each assignment $i \leftarrow i + c$ to a basic induction variable, create an assignment $j' \leftarrow j' + c \cdot b$
- Replace assignment to $j$ with $j \leftarrow j'$
- Initialize $j' \leftarrow a + i \cdot b$ at end of preheader
Example Strength Reduction
Induction Variables $j \sim (i, 0, 4)$ and $k \sim (i, a, 4)$

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\

  L_1 : & \text{ if } i \geq n \text{ goto } L_2 \\
  j & \leftarrow i \cdot 4 \\
  k & \leftarrow j + a \\
  x & \leftarrow M[k] \\
  s & \leftarrow s + x \\
  i & \leftarrow i + 1 \\

  \text{goto } L_1 \\

  L_2 \\

  \end{align*}
\]

before

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\
  j' & \leftarrow 0 \\
  k' & \leftarrow a \\

  L_1 : & \text{ if } i \geq n \text{ goto } L_2 \\
  j & \leftarrow j' \\
  k & \leftarrow k' \\
  x & \leftarrow M[k] \\
  s & \leftarrow s + x \\
  i & \leftarrow i + 1 \\
  j' & \leftarrow j' + 4 \\
  k' & \leftarrow k' + 4 \\

  \text{goto } L_1 \\

  L_2 \\

  \end{align*}
\]

after
Elimination

- Apply constant propagation, copy propagation, and dead code elimination
- Special case: elimination of induction variables that are
  - not used in the loop
  - only used in comparisons with loop-invariant variables
  - useless

**Useless variable**

A variable is **useless** in a loop $L$ if
- it is dead at all exits from $L$
- it is only used in its own definitions

**Example** After removal of $j$, $j'$ is useless
Rewriting Comparisons

Almost useless variable

A variable is almost useless in loop \( L \) if

- it is only used 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.

An almost useless variable can be made useless by rewriting the comparisons to use the related induction variable.
Coordinated induction variables

Let \( x \sim (i, a_x, b_x) \) and \( y \sim (i, a_y, b_y) \) be induction variables. \( x \) and \( y \) are coordinated if

\[
\frac{x - a_x}{b_x} = \frac{y - a_y}{b_y}
\]

throughout the execution of the loop, except during a sequence of statements of the form \( z_i \leftarrow z_i + c_i \) where \( c_i \) is loop-invariant.
Let $j \sim (i, a_j, b_j)$ and $k \sim (i, a_k, b_k)$ be coordinated induction variables.
Consider the comparison $k < n$ with $n$ loop-invariant.
Using $(j - a_j)/b_j = (k - a_k)/b_k$ the comparison can be rewritten as follows

\[ b_k(j - a_j)/b_j + a_k < n \]
\[ \Leftrightarrow \]
\[ b_k(j - a_j)/b_j < n - a_k \]
\[ \Leftrightarrow \]
\[
\begin{cases}
  j < (n - a_k)b_j/b_k + a_j & \text{if } b_j/b_k > 0 \\
  j > (n - a_k)b_j/b_k + a_j & \text{if } b_j/b_k < 0
\end{cases}
\]

where the right-hand sides are loop-invariant and their computation can be hoisted to the preheader.
Rewriting Comparisons

Restrictions

1. \((n - a_k)b_j\) must be a multiple of \(b_k\)
2. \(b_j\) and \(b_k\) must both be constants or loop invariants of known sign
Outline

1. Loops
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Safe programming languages check that the subscript is within the array bounds at each array operation.

Bounds for an array have the form $0 \leq i < N$ where $N > 0$ is the size of the array.

Implemented by $i <_u N$ (unsigned comparison).

Bounds checks redundant in well-written programs $\Rightarrow$ slowdown

For better performance: let the compiler prove which checks are redundant!

In general, this problem is undecidable.
Assumptions for Bounds Check Elimination in Loop $L$

1. There is an induction variable $j$ and loop-invariant $u$ used in statement $s_1$ of either of the forms
   - if $j < u$ goto $L_1$ else goto $L_2$
   - if $j \geq u$ goto $L_2$ else goto $L_1$
   - if $u > j$ goto $L_1$ else goto $L_2$
   - if $u \geq j$ goto $L_2$ else goto $L_1$

   where $L_2$ is out of the loop $L$.

2. There is a statement $s_2$ of the form
   - if $k <_u n$ goto $L_3$ else goto $L_4$

   where $k$ is an induction variable coordinated with $j$, $n$ is loop-invariant, and $s_1$ dominates $s_2$.

3. No loop nested within $L$ contains a definition of $k$.

4. $k$ increases when $j$ does: $b_j/b_k > 0$. 
Array-Bounds Checking

Objective

Insert test in preheader so that $0 \leq k < n$ in the loop.

Lower Bound

- Let $\Delta k_1, \ldots, \Delta k_m$ be the loop-invariant values added to $k$ inside the loop
- $k \geq 0$ everywhere in the loop if
  - $k \geq 0$ in the loop preheader
  - $\Delta k_1 \geq 0 \ldots \Delta k_m \geq 0$
Array-Bounds Checking

Upper Bound

- Let $\Delta k_1, \ldots, \Delta k_p$ be the set of loop-invariant values added to $k$ on any path between $s_1$ and $s_2$ that does not go through $s_1$.

- $k < n$ at $s_2$ if $k < n - (\Delta k_1 + \cdots + \Delta k_p)$ at $s_1$.

- From $(k - a_k)/b_k = (j - a_j)/b_j$ this test can be rewritten to $j < b_j/b_k(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$.

- It is sufficient that $u \leq b_j/b_k(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$ because the test $j < u$ dominates the test $k < n$.

- All parts of this test are loop-invariant!
Array-Bounds Checking Transformation

- Hoist loop-invariants out of the loop
- Copy the loop $L$ to a new loop $L'$ with header label $L'_h$
- Replace the statement “if $k < u \ n$ goto $L'_3$ else goto $L'_4$” by “goto $L'_3$”
- At the end of $L$’s preheader put statements equivalent to
  
  if $k \geq 0 \land \Delta k_1 \geq 0 \land \cdots \land \Delta k_m \geq 0$
  and $u \leq b_j/b_k(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$
  goto $L'_h$ else goto $L_h$
This condition can be evaluated at compile time if
1. all loop-invariants in the condition are constants; or
2. \( n \) and \( u \) are the same temporary, \( a_k = a_j, b_k = b_j \) and no \( \Delta k \)'s are added to \( k \) between \( s_1 \) and \( s_2 \).

The second case arises for instance with code like this:

```java
int u = a.length;
int i = 0;
while (i < u) {
    sum += a[i];
    i++;
}
```

assuming common subexpression elimination for \( a.length \)

Compile-time evaluation of the condition means to unconditionally use \( L \) or \( L' \) and delete the other loop

Clean up with elimination of unreachable and dead code
Array-Bounds Checking Generalization

- Comparison of $j \leq u'$ instead of $j < u$
- Loop exit test at end of loop body: A test
  - $s_2 : \text{if } j < u \text{ goto } L_1 \text{ else goto } L_2$
  
  where $L_2$ is out of the loop and $s_2$ dominates all loop back edges; the $\Delta k_i$ are between $s_2$ and any back edge and between the loop header and $s_1$
- Handle the case $b_j/b_k < 0$
- Handle the case where $j$ counts downward and the loop exit tests for $j \geq l$ (a loop-invariant lower bound)
- The increments to the induction variable may be "undisciplined" with non-obvious increment:

```java
1   while (i<n-1) { 
2      if (sum<0) { i++; sum += i; i++ } else { i += 2; } 
3      sum += a[i];
4   }
```
Loop Unrolling

- For loops with small body, some time is spent incrementing the loop counter and testing the exit condition.
- **Loop unrolling** optimizes this situation by putting more than one copy of the loop body in the loop.

To unroll a loop $L$ with header $h$ and back edges $s_i \rightarrow h$:

1. Copy $L$ to a new loop $L'$ with header $h'$ and back edges $s'_i \rightarrow h'$.
2. Change the back edges in $L$ from $s_i \rightarrow h$ to $s_i \rightarrow h'$.
3. Change the back edges in $L'$ from $s'_i \rightarrow h'$ to $s'_i \rightarrow h$. 
Loop Unrolling Example (Still Useless)

\[\begin{align*}
L_1 : & \quad x \leftarrow M[i] \\
& \quad s \leftarrow s + x \\
& \quad i \leftarrow i + 4 \\
& \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \\
L_2 & \\
\end{align*}\]

before

\[\begin{align*}
L_1 : & \quad x \leftarrow M[i] \\
& \quad s \leftarrow s + x \\
& \quad i \leftarrow i + 4 \\
& \text{if } i < n \text{ goto } L' \text{ else } L_2 \\
L' & \\
\end{align*}\]

after
No gain, yet

Needed: induction variable \( i \) such that every increment
\[ i \leftarrow i + \Delta \]
dominates every back edge of the loop
\[ \Rightarrow \]
each iteration increments \( i \) by the sum of the \( \Delta \)s
\[ \Rightarrow \]
increments and tests can be moved to the back edges of loop

In general, a separate epilogue is needed to cover the remaining iterations because a loop that is unrolled \( K \) times can only do multiple-of-\( K \) iterations.
Loop Unrolling Example

\[ L_1 : \quad x \leftarrow M[i] \]
\[ s \leftarrow s + x \]
\[ x \leftarrow M[i + 4] \]
\[ s \leftarrow s + x \]
\[ i \leftarrow i + 8 \]
if \( i < n \) goto \( L_1 \) else \( L_2 \)

\[ L_2 \]

only even numbers

\[ L_1 : \quad x \leftarrow M[i] \]
\[ s \leftarrow s + x \]
\[ x \leftarrow M[i + 4] \]
\[ s \leftarrow s + x \]
\[ i \leftarrow i + 8 \]
if \( i < n - 4 \) goto \( L_1 \) else \( L_2 \)

\[ L_2 : \quad x \leftarrow M[i] \]
\[ s \leftarrow s + x \]
\[ i \leftarrow i + 4 \]
\[ L_3 \]

\[ L_{2'} : \quad \text{if } i < n \text{ goto } L_2 \text{ else } L_3 \]

\[ L_2 : \quad \text{with epilogue} \]