Contents of Lecture 8

- Purpose of Operator Strength Reduction, OSR
- Original algorithm for OSR
- The SSA Graph
- Strongly Connected Components
- Tarjan’s Algorithm for computing the strongly connected components
- OSR on SSA Form
double a[N];
for (i = 0; i < N; ++i)
    x += a[i];

double* p = a;
double* end = &a[N];
while (p < end)
    x += *p++;

- The most important purpose is to rewrite the code to the left into the code to the right.
- C/C++ compilers are required to make it possible to use the address of the array element after the last declared element.
- Typically, in total one extra byte might be wasted in memory due to this.
- It’s **not** one extra byte per array but rather per memory segment.
double a[N];

double* p = &a[N];

for (i = N-1; i >= 0; --i)
    x += a[i];

while (--p >= a)
    x += *p;

- In the last iteration p == a[-1] in the comparison.
- The compiler is not required to make that address valid.
- The code to the right triggers undefined behavior.
OSR is also known as Induction Variable Elimination

do {
    x = x + a[i];
    i = i + 1;
} while (i < N);

do {
    s = i * 4;
    t = load a+s;
    x = x + t;
    i = i + 1;
} while (i < N);
The primary goal is to get rid of the multiplication

\[
\text{do } \{ \\
\quad s = i \times 4; \\
\quad t = \text{load a}+s; \\
\quad x = x + t; \\
\quad i = i + 1; \\
\} \text{ while (i < N);} 
\]

- \(i\) is a \textit{basic} induction variable
- Classes of \textit{dependent} induction variables: \(j \leftarrow b \times i + c, i\) is a basic IV
- \(s \leftarrow 4 \times i + 0\)
Strength reduction

\[
\text{do } \{ \\
\quad s = i \times 4; \\
\quad t = \text{load } a+s; \\
\quad x = x + t; \\
\quad i = i + 1; \\
\} \text{ while (} i < N); \]

\[
\text{s = 4 * i; } \text{do } \{ \\
\quad t = \text{load } a+s; \\
\quad x = x + t; \\
\quad i = i + 1; \\
\quad s = s + 4; \\
\} \text{ while (} i < N); \]

- Initialize the dependent IV before the loop
- Increment the dependent IV just after the basic IV is incremented
- Maybe we can get rid of the basic IV now?
Linear function test replacement

\[ s = 4 * i; \]
\[ \text{do } \{ \]
\[ \quad t = \text{load a+s}; \]
\[ \quad x = x + t; \]
\[ \quad i = i + 1; \]
\[ \quad s = s + 4; \]
\[ \} \text{ while (i < N);} \]
\[ m = 4 * N; \]
\[ s = 4 * i; \]
\[ \text{do } \{ \]
\[ \quad t = \text{load a+s}; \]
\[ \quad x = x + t; \]
\[ \quad s = s + 4; \]
\[ \} \text{ while (s < m);} \]

- \[ s = i \times b + c \] (we have \( b = 4 \) and \( c = 0 \))
- \[ i = \frac{s-c}{b} \]
- \[ i < N \Rightarrow \frac{s-c}{b} < N \Rightarrow s < N \times b + c, \text{ if } b > 0 \]
procedure operator_strength_reduce(ssa_graph)
  dfnum ← 0
  empty stack
  for each vertex $v \in ssa_{\text{graph}}$ do
    visited$(v) \leftarrow$ false
  for each vertex $v \in ssa_{\text{graph}}$ do
    if (not visited$(v)$)
      strong_connect$(v)$
  end
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum +1
    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))
    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
int dfnum  /* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum +1

  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))
  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
end
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))
    
if (lowlink(v) = dfn(v))
    scc ← ∅
    do
        w ← pop()
        add w to scc
    while (w ≠ v)
process_scc(scc)
int dfnum

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
    end
int dfnum

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
    end
procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
Tarjan’s Algorithm: Initial Processing of 6

```plaintext
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
    end
```

```
0, 0
1, 1
2, 2
3, 3
4, 4
5, 5
6, 6
```

stack

```
0
1
2
3
4
5
6
```
(6, 2) ⇒ 6 in same scc as 2.

```
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
```
(6, 3). no action.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()  
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
6 remains on the stack.

```c
int    dfnum          /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
New lowlink and remains.

```plaintext
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum + 1

  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))

  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
end
```
Tarjan’s Algorithm: More Processing of 4

- New lowlink and remains.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
```

---

stack

0 0
1 1
2 2
3 3
4 2
5 2
6 2

0
1
2
3
4
5
6

jonasskeppstedt.net Lecture 8 2020 20/48

int  dfnum
     /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
Tarjan’s Algorithm: Processing of 7

- Lowlink is set.

```c
int dfnum  /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1
    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))
    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
    process_scc(scc)
```
Tarjan’s Algorithm: More Processing of 2

- Remove SCC from stack

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum +1

  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))

  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
  end
```
No path from 2 to 8.

```plaintext
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```
8 is its own SCC.

```plaintext
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
```
1 is its own SCC.

```plaintext
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
  dfn(v) ← dfnum
  lowlink(v) ← dfnum
  visited(v) ← true
  push(v)
  dfnum ← dfnum + 1

  for each w ∈ succ(v) do /* operands(v) = succ(v) */
    if (not visited(w)) {
      strong_connect(w)
      lowlink(v) ← min(lowlink(v), lowlink(w))
    } else if (dfn(w) < dfn(v) and w is on stack)
      lowlink(v) ← min(lowlink(v), dfn(w))

  if (lowlink(v) = dfn(v))
    scc ← ∅
    do
      w ← pop()
      add w to scc
    while (w ≠ v)
    process_scc(scc)
```
Tarjan’s Algorithm: More Processing of 0

- 0 is its own SCC.

```c
int dfnum /* Depth-first search number. */

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do /* operands(v) = succ(v) */
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v) and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```

stack
Consider the edge \((v, w)\).

When \(w\) is not yet visited we must visit it by calling
\[\text{strong\_connect}(w)\].

If \(w\) has been visited, we have two main cases:

1. \(w\) is not on the stack, because it has already found its SCC.
2. \(w\) is on the stack, because it’s waiting for being popped.
   - If \(\text{dfn}(w) < \text{dfn}(v)\) then \(v\) must set its lowlink so it does not think it is its own SCC.
   - If \(\text{dfn}(w) \geq \text{dfn}(v)\) then no more information for \(v\) is available. There is another path from \(v\) to \(w\) due to which they will belong to the same SCC.
double a[N];
for (i = 0; i < N; ++i)
    x += a[i];
We first find all strongly connected components of the SSA graph.

We want to copy the SCC of $i$ and modify the copy for $t_1$.

Therefore we want to have processed $i$ before processing $t_1$.

Let us start with $x$. 
Processing of $x_0$

- $\text{SCC}_0 = \{x_0\}$. Empty stack.
- Nodes processed in a SCC are green.
- Next processing $x_1$. 

 jonasskeppstedt.net  Lecture 8  2020  31 / 48
Processing of $x_1$ and $x_2$

- $x_1$ and $x_2$ are pushed and then the search continues with $t_2$.
- Nodes on the stack are red.
- Next processing $t_2$. 
Processing of $t_2$

- $x_0 \rightarrow 0$
- $x_1 \rightarrow \phi(x_0, x_2)$
- $x_2 \rightarrow x_1 + t_2$
- $t_1 \rightarrow i_1 \times 8$
- $M[a + t_1]$
- $i_0 \rightarrow 0$
- $i_1 \rightarrow \phi(i_0, i_2)$
- $i_2 \rightarrow i_1 + 1$

Next processing $t_1$. 
Processing of $t_1$

- Next processing $i_2$. 
Processing of $i_2$ and $i_1$

Next processing $i_0$. 
Processing of $i_0$

$\phi(x_0, x_2) \rightarrow x_1 + t_2$

$t_1 \rightarrow i_1 \times 8$

$t_2 \rightarrow M[a + t_1]$

$i_0 \rightarrow \phi(i_0, i_2)$

$x_0 \rightarrow 0$

$x_1 \rightarrow \phi(x_0, x_2)$

$x_2 \rightarrow x_1 + t_2$

$i_1 \rightarrow i_1 + 1$

$i_2 \rightarrow i_1 + 1$

$SCC_1 = \{i_0\}$

Next more processing in $i_2$. 
Classifying $\text{SCC}_2 = \{i_1, i_2\}$

- $\text{SCC}_2 = \{i_1, i_2\}$
- $\text{SCC}_2$ is an **induction variable** due it consists of a $\phi$-function and an add with a **region constant**.
- A region constant is not modified in a loop, i.e. it’s a number or its definition strictly dominates the loop header.
Replacing $i_1 \times 8$

- $SCC_3 = \{ t_1 \}$
- $SCC_3$ is a multiplication of an induction variable and a region constant.
- Therefore $SCC_3$ is replaced by a modified copy of $SCC_2$ with $\phi(i)$. 
Due to the replacement, the assignment to $t_1$ becomes dead code. There is a very beautiful algorithm to remove $t_1$ and other dead code that we will look at during the next lecture.
Also \( a + t_1 \) can be Replaced

- Due to Tarjan’s algorithm we can start in any node and be sure we have already processed the operand nodes, when a variable’s definition is going to be replaced.
- Not only multiplications but also some additions can be replaced, but we don’t show this in the example.
When nodes have been popped from the stack and collected in a SCC, the following is performed.

A SCC has the attribute **header** which is the header of a loop in the control flow graph.

```java
procedure process_scc(scc)
    if (scc has a single member n)
        if (valid_form(n))
            replace(n, iv, rc)
        else
            header(n) ← ⊥
    else
        classify(scc)
end
```
Valid Forms of Definition for Replacement

- $iv$ is induction variable
- $rc$ is region constant

```python
function valid_form(n)
    if (n is of form $x \leftarrow iv \times rc$
        or $n$ is of form $x \leftarrow rc \times iv$
        or $n$ is of form $x \leftarrow iv \pm rc$
        or $n$ is of form $x \leftarrow rc + iv$)
        return true
    else
        return false
end
```
Definition of Region Constant

```c
function region_const(x, header)
    return x is constant or vertex(x) strictly dominates header
end
```

```c
scanf("%d %d", &a, &b);
while (i < n) {
    x += u[a * i + b];
    i += 1;
}
```

- The variables a and b are region constants in the loop.
Reverse Post Order

```plaintext
int i

procedure dfs(v)
    visited(v) ← true
    for each w ∈ succ(v) do
        if (not visited(w))
            dfs(w)
    i ← i - 1
    rpo(v) ← i
end

procedure compute_rpo(CFG )
    i ← |V|
    for each vertex v do
        visited(v) ← false
        dfs(v)
end
```
procedure classify(scc)
    for each $n \in scc$ do
        if $\text{rpo}(\text{vertex}(n)) < \text{rpo}(\text{header})$
            $\text{header} \leftarrow \text{vertex}(n)$
    for each $n \in scc$ do
        if $(\text{operator}(n) \notin \{\phi, +, -, \text{move}\})$
            $scc$ is not an induction variable
        else
            for each operand $\omega \in \text{operands}(n)$ do
                if $(\omega \notin scc \text{ and not } \text{region}_\text{const}(\omega, \text{header}))$
                    $scc$ is not an induction variable
            if ($scc$ is an induction variable)
                for each $n \in scc$ do
                    $\text{header}(n) \leftarrow \text{header}$
            else
                for each $n \in scc$ do
                    if $(\text{valid}_\text{form}(n))$
                        replace$(n, iv, rc)$
                    else
                        $\text{header}(n) \leftarrow \bot$
        end
procedure replace(operation, iv, rc)
    result ← reduce(opcode(operation), iv, rc)
    replace operation with mov using result as source
    header(operation) ← header(iv)
end
function reduce(operation, iv, rc)
    result ← lookup(opcode, iv, rc)
    if (result is not found)
        result ← new_temp()
        install(opcode, iv, rc, result)
        new_def ← copy_def(iv, result)
        for each operand ω in new_def do
            if (ω is an induction variable)
                replace ω with reduce(opcode, ω, rc)
            else if (opcode = × or new_def is a ϕ)
                replace ω with apply(opcode, ω, rc)
        return result
    end
function apply(opcode, op1, op2)
    result ← lookup(opcode, op1, op2)
    if (result is not found)
        if (op1 is an induction variable and op2 is a region constant)
            result ← reduce(opcode, op1, op2)
        else if (op2 is an induction variable and op1 is a region constant)
            result ← reduce(opcode, op2, op1)
        else
            result ← new_temp()
            install(opcode, op1, op2, result)
    choose the location where the operation will be inserted
    perform constant folding if possible
    create a new operation at the chosen location
    return result
end