Contents of Lecture 6

- Copy Propagation during Translation to SSA Form
- Hash-Based Value Numbering during Translation to SSA Form
- Global Value Numbering on SSA Form
Copy Propagation During Translation to SSA Form

\[
\begin{align*}
    a_0 & \leftarrow x + y \\
    b_0 & \leftarrow a_0 \\
    c_0 & \leftarrow z + 44 \\
    d_0 & \leftarrow b_0 + c_0
\end{align*}
\]

- Instead of pushing \( b_0 \) on \( b \)'s stack...
- we can push \( a_0 \) on \( b \)'s stack
- You will implement this optimization in Lab 2.
The name is due to each expression, e.g. \( t_i \leftarrow a + b \), is given a number, essentially a hash-table index.

In subsequent occurrences of \( t_j \leftarrow a + b \) it is checked whether the statement can be changed to \( t_j \leftarrow t_i \).

This is a very old optimization technique with one version that is performed during translation to SSA Form and other versions when the code already is on SSA Form.

There are obviously older versions used before SSA Form but we will not look at them.
An expression $a + b$ is **redundant** if it is evaluated multiple times with identical values of the operands.

Eliminating redundant expressions is a very important optimization goal.

There are different approaches to redundancy elimination, including:

1. Hash-Based Value Numbering
2. Global Value Numbering
3. Common Subexpression Elimination
4. Code Motion out of Loops
5. Partial Redundancy Elimination

We will study 1, 2, and 5 in detail.
Some Optimization Goals

- Redundancy Elimination
- Operator Strength Reduction – replace slow instructions with faster
- Control Flow Optimization
  - For example place basic blocks in an order which reduces the number of dynamically executed branches.
  - Inline functions to eliminate function call overhead.
- Memory Hierarchy Optimization
  - Register allocation
  - Locality optimizations using e.g. tiling.
  - Cut functions into two halves and put hot halves together in cache blocks or virtual pages.
- Pipeline Optimizations through instruction scheduling.
- Parallelization
  - SIMD
  - Multicore
In vertex 1 the expression $a_0 + b_0$ is first computed.

- The redundant occurrences of $a_0 + b_0$ can easily be removed.
- On SSA Form we simply check that the variable versions are the same in the current and previous occurrence.
Example 2

The occurrences in vertices 3 and 4 cannot mistakenly be regarded as useful due to mismatching variable versions.
Obviously there are no redundant expressions here.

We could perhaps save memory by computing $a_0 + b_0$ in vertex 1 but that is not a goal for redundancy elimination.

Which data structure should we use for performing value numbering during translation to SSA Form?
procedure rename(w)
  oldLHS ← empty list
  enter new scope in hash table
  for each statement t in w do
    for each variable V ∈ RHS(t)
      replace use of V by use of V_i where i = top(S(V))

  V = LHS(t)
  if (V = null)
    continue
  add V to oldLHS
simplify \( t \) using e.g. \( V_i - V_i = 0 \)

\( h \leftarrow \text{lookup } RHS(t) \text{ in hash table} \)

\textbf{if} (\( h \) was found)

\hspace*{1em} push left-hand side of \( h \) onto \( S(V) \)

\textbf{else} \{ 

\hspace*{1em} \( i \leftarrow C(V) \)

\hspace*{1em} replace \( V \) by \( V_i \)

\hspace*{1em} push \( i \) onto \( S(V) \)

\hspace*{1em} \( C(V) \leftarrow C(V) + 1 \)

\hspace*{1em} install \( t \) in hash table

\}


for each $v \in \text{succ}(w)$ do
    $j \leftarrow \text{which}\_\text{pred}(w, v)$
    for each $\phi$-function in $v$ do
        replace the $j$-th operand in $RHS(\phi)$ by $V_i$ where $i = \text{top}(S(V))$
for each $v \in \text{children}(w)$ do
    rename($v$)
for each variable $V$ in oldLHS do
    pop($V$)
exit scope in hash table
Global Value Numbering (GVN)

- Global Value Numbering was one of the first optimization presented on SSA Form, and was invented by IBM Research.
- SSA Form was explained in that paper as well to introduce this novelty to the reader.
- IBM actually uses an unpublished version of this algorithm which is better.
- Mårten Kongstad, D99, implemented this algorithm in his Master’s Thesis as a pass in GCC and observed performance improvements of up to 6.1%.
Recall that in constant propagation only the start vertex is initially assumed to be executable.

In GVN the initial assumption is that all instructions with the same operation will produce the same value.

I.e. all adds produce the same value, etc.

This most likely is not the case, of course.

Then, for example, the add instructions are inspected to check whether the compiler can determine that two such instructions do not produce the same value.

When the algorithm terminates, it has proved which instructions produce the same value.
The set $I$ of all instructions in a control flow graph is partitioned into blocks $B_j$, i.e. $\bigcup B_j = I$.

Initially a block $B_j$ consists of all instructions with the same operator and type.

Each instruction $i$ has a number of operands.

Two instructions are regarded as equivalent if they belong to the same block $B_j$ and their operands come from the same blocks.

A variable $x$ with unknown value is put in a singleton block $B_x$.

$\phi$-functions can be equal only if they belong to the same basic block.
An Example of Equivalent Instructions

- We denote an instruction with the variable defined by it, and $\text{left}(a)$ and $\text{right}(a)$ denote the instruction which define the left and right operand of instruction $a$, respectively.

\begin{align*}
a &= x + y; \\
b &= x - z; \\
c &= x + z; \\
d &= a - b; \\
e &= a + d; \\
f &= a + b; \\
g &= b + d;
\end{align*}

- $B_x = \{x\}$, $B_y = \{y\}$, and $B_z = \{z\}$
- $B_+ = \{a, c, e, f, g\}$
- $B_- = \{b, d\}$

Let us check some instructions:

- $\text{left}(e) \in B_+$ and $\text{left}(f) \in B_+$ and $\text{right}(e) \in B_-$ and $\text{right}(f) \in B_-$ so we still think $e \equiv f$.
- $\text{left}(f) \in B_+$ and $\text{left}(g) \in B_-$ so $f \not\equiv g$.

What should we do when we have discovered that two instructions from the same block cannot be equivalent?
We just discovered that $f \neq g$.

Therefore we split $B_+$ into $B'_+$ and $B''_+$.

$B_x = \{x\}$, $B_y = \{y\}$, and $B_z = \{z\}

$B'_+ = \{a, c, e, f\}$

$B''_+ = \{g\}$

$B_- = \{b, d\}$

Thus, we split blocks when we discover that two members cannot be equivalent due to their respective operands come from different blocks.

How should we practically perform the splitting?
The Basic Algorithm

procedure $N^2$-partition
let $\pi_0 = \{B_0, B_1, B_2, \ldots, B_p\}$ be the initial partition
$i \leftarrow 0$
$change \leftarrow true$
while ($change$) do
  $change \leftarrow false$
  $k \leftarrow 0$
  for each $B_j \in \pi_i$ do
    take one node $v$ from $B_j$
    create a new block $B_k$ in $\pi_{i+1}$
    put $v$ in $B_k$ in $\pi_{i+1}$
    for each node $w \in B_j$ do
      if (match($v$, $w$))
        add $w$ to $B_k$ in $\pi_{i+1}$
      else {
        if ($B_{k+1}$ has not already been created)
          create a new block $B_{k+1}$ in $\pi_{i+1}$
          $change \leftarrow true$
        add $w$ to $B_{k+1}$ in $\pi_{i+1}$
      }
  end
All instructions in a block $B_k$ in the final partition $\pi$ produce the same value.

Suppose $a$ and $b$ are members of $B_k$.

Using dominance at the instruction level, if $a \gg b$ then $b$ is redundant and can be replaced with $a$. 
Assume we have the statement:
\[ b = a[0] + a[1] + a[2] + \ldots + a[n]; \]
All the add instructions will belong to the same block in \( \pi_0 \).
Then one instruction is removed each iteration which results in an \( N^2 \) algorithm.
This algorithm is too slow in practice and we will next look at a faster algorithm.
The main problem with the \( N^2 \)-algorithm is that a block is used to split itself by inspecting all its members.
A Key Idea for a Faster Algorithm

- Instead of taking one block $B_k$ and inspect every instruction in it, and either putting it in $B'_k$ or in $B''_k$ we can take one block and use it to split other blocks. Each block is given a sword.

- To simplify the description let us for the moment only consider unary operators.

- Consider blocks $B_i$ and $B_j$ and whether $B_j$ should be split due to $B_i$.

- Assume some of the members of $B_j$ take their operand from $B_i$ while some others don’t.

- Then $B_j$ must be split, and those with operands from $B_i$ should be put in $B'_j$ and the rest in $B''_j$.

- $INV(B_i)$ is the set of instructions which take their operand from $B_i$.

- If $INV(B_i) \cap B_j \neq \emptyset$ and $B_j \notin INV(B_i)$ then $B_j$ must be split.
Consider a block $B_j$ with instructions.

Let $x \in B_j$ and assume $x$ only operand was defined in $B_i$.

We write this as $f(x) \in B_i$. $INV(B_i) = \{ v \mid f(v) \in B_i \}$.

We might split $B_j$ due to $B_i$ into $B'_j$ and $B''_j$.

$B'_j = \{ v \mid v \in B_j \land f(v) \in B_i \}$ and $B''_j = \{ v \mid v \in B_j \land f(v) /\in B_i \}$.

Recall $B_j$ is split due to $B_i$ if $INV(B_i) \cap B_j \neq \emptyset$ and $B_j \not\subseteq INV(B_i)$

If $INV(B_i) \cap B_j = \emptyset$ then $B_j$ is completely unrelated to $B_i$, and $B_j$ should therefore not be split due to $B_i$.

If $B_j \subseteq INV(B_i)$ then all instructions in $B_j$ take their operand from $B_i$, and $B_j$ should therefore not be split due to $B_i$. 
Every initial block $B_j$ is put on a worklist $W$.

$B_i$ is taken from $W$ and $INV(B_i)$ is computed.

Then all other blocks are inspected and split if:

$INV(B_i) \cap B_j \neq \emptyset$ and $B_j \not\subseteq INV(B_i)$

A block $B_i$ on the worklist $W$ is equipped with a sword to cut all other blocks into two pieces.

If a split block $B_j$ also was on $W$, then both pieces of $B_j$, i.e. $B_j'$ and $B_j''$, will remain on $W$ (but not $B_j$ obviously).

What should we do if $B_j$ is not on the worklist?
Splitting a Block $B_j$ not on the Worklist

- Assume $B_j \notin W$.
- It means $B_j$ already has had its chance to cut other blocks.
- However, when $B_j$ is split into $B_j'$ and $B_j''$ some block $B_k$ might now have to be split!
- Do we have to put both $B_j'$ and $B_j''$ into $W$?
- Assume that $\forall v \in B_k \ f(v) \in B_j$. Then, if $B_j$ is split into $B_j'$ and $B_j''$, we must either have $f(v) \in B_j'$ or $f(v) \in B_j''$, hence $f(v) \in B_j' \iff f(v) \notin B_j''$. Therefore, to split $B_k$ we can use either $B_j'$ or $B_j''$:

$$\{ \ v \mid v \in B_k \wedge f(v) \in B_j' \} = B_k - \{ \ v \mid v \in B_k \wedge f(v) \in B_j'' \}$$
$$\{ \ v \mid v \in B_k \wedge f(v) \in B_j'' \} = B_k - \{ \ v \mid v \in B_k \wedge f(v) \in B_j' \}$$

- By using the smaller of the sets $B_j'$ and $B_j''$ we can achieve a time complexity of $O(N\log N)$ where $N$ is the number of nodes in the value graph.
int h(int a, int b) {
    int x, y;
    x = 1;
    y = 1;
    do {
        a = a + b;
        x = x + a;
        y = y + a;
    } while (a > 0);
    return x + y;
}

int h(int a, int b) {
    int x, y;
    x = 1;
    do {
        a = a + b;
        x = x + a;
    } while (a > 0);
    return x + x;
}