Theorem

Assume that every program point is reachable from `start` and the program is in SSA form without assignments to dead variables.

Let $\lambda$ denote the maximal number of simultaneously live variables and $G$ the interference graph of the program variables. Then:

$$\lambda = \omega(G) = \chi(G)$$

where $\omega(G), \chi(G)$ are the maximal size of a clique in $G$ and the minimal number of colors for $G$, respectively.

A minimal coloring of $G$, i.e., an optimal register allocation can be found in polynomial time.
Discussion

• By the theorem, the number $\lambda$ of required registers can be easily computed :-)

• Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers!

• Thus here, we may, e.g., insist on keeping iteration variables from inner loops.
Discussion

• By the theorem, the number $\lambda$ of required registers can be easily computed :-)

• Thus variables which are to be spilled to memory, can be determined ahead of the subsequent assignment of registers !

• Thus here, we may, e.g., insist on keeping iteration variables from inner loops.

• Clearly, always $\lambda \leq \omega(G) \leq \chi(G')$ :-)

Therefore, it suffices to color the interference graph with $\lambda$ colors.

• Instead, we provide an algorithm which directly operates on the cfg ...

628
Observation

- Live ranges of variables in programs in SSA form behave similar to live ranges in basic blocks!
- Consider some dfs spanning tree $T$ of the cfg with root $\text{start}$.
- For each variable $x$, the live range $L[x]$ forms a tree fragment of $T$!
- A tree fragment is a subtree from which some subtrees have been removed...
Example

\[ x = M[i]; \]
\[ y = 1; \]
\[ \text{Neg}(x > 1) \]
\[ M[a] = y; \]
\[ x = x - 1; \]
\[ \text{Pos}(x > 1) \]
\[ y = x * y; \]
Discussion

- Although the example program is not in SSA form, all live ranges still form tree fragments  :-)  
- The intersection of tree fragments is again a tree fragment !
- A set $C$ of tree fragments forms a clique iff their intersection is non-empty !!!
- The greedy algorithm will find an optimal coloring ...
Proof of the Intersection Property

(1) Assume \( I_1 \cap I_2 \neq \emptyset \) and \( v_i \) is the root of \( I_i \). Then:

\[
v_1 \in I_2 \quad \text{or} \quad v_2 \in I_1
\]

(2) Let \( C \) denote a clique of tree fragments. Then there is an enumeration \( C = \{ I_1, \ldots, I_r \} \) with roots \( v_1, \ldots, v_r \) such that

\[
v_i \in I_j \quad \text{for all} \quad j \leq i
\]

In particular, \( v_r \in I_i \) for all \( i \). :-)}
The Greedy Algorithm

forall \((u \in \text{Nodes})\) \(\text{visited}[u] = \text{false}\);
forall \((x \in \mathcal{L}[\text{start}])\) \(\Gamma(x) = \text{extract}(\text{free})\);
alloc(\text{start});

\begin{verbatim}
void alloc (Node u) {
    \text{visited}[u] = \text{true};
    forall ((lab, v) \in \text{edges}[u])
        if (\neg \text{visited}[v]) {
            forall (x \in \mathcal{L}[u] \setminus \mathcal{L}[v]) \text{insert}(\text{free}, \Gamma(x));
            forall (x \in \mathcal{L}[v] \setminus \mathcal{L}[u]) \Gamma(x) = \text{extract}(\text{free});
            alloc(v);
        }
}
\end{verbatim}
Example

0
read();
1
x = M[A];
2
y = x + 1;
3
Neg(y) Pos(y)
4
t = -y \cdot y;
5
M[A] = t;
6
z = x \cdot x
7
M[A] = z;
8
Example

read();

1  
\( x = M[A]; \)

2  
\( y = x + 1; \)

3  
\( \text{Neg (} y \text{)} \)

4  
\( t = -y \cdot y; \)

5  
\( M[A] = t; \)

6  
\( \text{Pos (} y \text{)} \)

7  
\( z = x \cdot x \)

8  
\( M[A] = z; \)

read();

1  
\( R_1 = M[A]; \)

2  
\( R_2 = R_1 + 1; \)

3  
\( \text{Neg (} R_2 \text{)} \)

4  
\( R_1 = -R_2 \cdot R_2; \)

5  
\( M[A] = R_1; \)

6  
\( \text{Pos (} R_2 \text{)} \)

7  
\( R_1 = R_1 \cdot R_1 \)

8  
\( M[A] = R_1; \)
Remark:

- Intersection graphs for tree fragments are also known as cordal graphs ...
- A cordal graph is an undirected graph where every cycle with more than three nodes contains a cord :-)
- Cordal graphs are another sub-class of perfect graphs :-))
- Cheap register allocation comes at a price:

  when transforming into SSA form, we have introduced parallel register-register moves :-(
Problem

The parallel register assignment:

\[ \psi_1 = R_1 = R_2 \mid R_2 = R_1 \]

is meant to exchange the registers \( R_1 \) and \( R_2 \) :-)

There are at least two ways of implementing this exchange ...
Problem

The parallel register assignment:

\[ \psi_1 = R_1 = R_2 \mid R_2 = R_1 \]

is meant to exchange the registers \( R_1 \) and \( R_2 \) :-)

There are at least two ways of implementing this exchange ...

(1) Using an auxiliary register:

\[
\begin{align*}
R &= R_1; \\
R_1 &= R_2; \\
R_2 &= R;
\end{align*}
\]
(2) **XOR:**

\[
\begin{align*}
R_1 &= R_1 \oplus R_2; \\
R_2 &= R_1 \oplus R_2; \\
R_1 &= R_1 \oplus R_2;
\end{align*}
\]
(2) XOR:

\[
R_1 = R_1 \oplus R_2; \\
R_2 = R_1 \oplus R_2; \\
R_1 = R_1 \oplus R_2;
\]

But what about cyclic shifts such as:

\[
\psi_k = R_1 = R_2 \mid \ldots \mid R_{k-1} = R_k \mid R_k = R_1
\]

for \( k > 2 \) ??
(2) **XOR:**

\[
R_1 = R_1 \oplus R_2; \\
R_2 = R_1 \oplus R_2; \\
R_1 = R_1 \oplus R_2;
\]

But what about cyclic shifts such as:

\[
\psi_k = R_1 = R_2 \mid \ldots \mid R_{k-1} = R_k \mid R_k = R_1
\]

for \( k > 2 \) ??

Then at most \( k - 1 \) swaps of two registers are needed:

\[
\psi_k = R_1 \leftrightarrow R_2; \\
R_2 \leftrightarrow R_3; \\
\ldots \\
R_{k-1} \leftrightarrow R_k;
\]
Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts 😄
- Any permutation of $n$ registers with $r$ shifts can be realized by $n - r$ swaps ...
Next complicated case: permutations.

- Every permutation can be decomposed into a set of disjoint shifts
  :-) 

- Any permutation of \( n \) registers with \( r \) shifts can be realized by \( n - r \) swaps ...

**Example**

\[ \psi = R_1 = R_2 \mid R_2 = R_5 \mid R_3 = R_4 \mid R_4 = R_3 \mid R_5 = R_1 \]

consists of the cycles \((R_1, R_2, R_5)\) and \((R_3, R_4)\). Therefore:

\[ \psi = R_1 \leftrightarrow R_2; \]
\[ R_2 \leftrightarrow R_5; \]
\[ R_3 \leftrightarrow R_4; \]
The general case:

- Every register receives its value at most once.
- The assignment therefore can be decomposed into a permutation together with tree-like assignments (directed towards the leaves) ...

Example

\[ \psi = R_1 = R_2 \mid R_2 = R_4 \mid R_3 = R_5 \mid R_5 = R_3 \]

The parallel assignment realizes the linear register moves for \( R_1 \), \( R_2 \) and \( R_4 \) together with the cyclic shift for \( R_3 \) and \( R_5 \):

\[
\psi = R_1 = R_2; \\
R_2 = R_4; \\
R_3 \leftrightarrow R_5; \\
\]

644
Interprocedural Register Allocation:

→ For every local variable, there is an entry in the stack frame.
→ Before calling a function, the locals must be saved into the stack frame and be restored after the call.
→ Sometimes there is hardware support :-) Then the call is transparent for all registers.
→ If it is our responsibility to save and restore, we may ...
  • save only registers which are over-written :-) 
  • restore overwritten registers only.
→ Alternatively, we save only registers which are still live after the call — and then possibly into different registers ➞ reduction of life ranges :-)
3.2 Instruction Level Parallelism

Modern processors do not execute one instruction after the other strictly sequentially.

Here, we consider two approaches:

(1) VLIW (Very Large Instruction Words)
(2) Pipelining
VLIW:

One instruction simultaneously executes up to $k$ (e.g., 4) elementary Instructions.

Pipelining:

Instruction execution may overlap.

Example:

$$w = (R_1 = R_2 + R_3 \mid D = D_1 \times D_2 \mid R_3 = M[R_4])$$
Warning:

- Instructions occupy hardware resources.
- Instructions may access the same busses/registers \implies hazards
- Results of an instruction may be available only after some delay.
- During execution, different parts of the hardware are involved:

  [Diagram showing Fetch, Decode, Execute, Write]

- During \textbf{Execute} and \textbf{Write} different internal registers/busses/alu's may be used.
We conclude:

Distributing the instruction sequence into sequences of words is amenable to various constraints ...

In the following, we ignore the phases Fetch und Decode :-)

Examples for Constraints:

(1) at most one load/store per word;
(2) at most one jump;
(3) at most one write into the same register.
Example Timing:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point Operation</td>
<td>3</td>
</tr>
<tr>
<td>Load/Store</td>
<td>2</td>
</tr>
<tr>
<td>Integer Arithmetic</td>
<td>1</td>
</tr>
</tbody>
</table>

Timing Diagram:

\[ R_1 \quad R_2 \quad R_3 \quad D \]

\[ \begin{array}{cccc}
0 & 5 & -1 & 2 & 0.3 \\
1 & 1 & & & \\
2 & & & 49 & \\
3 & & & & 17.4
\end{array} \]

*R_3* is over-written, after the addition has fetched 2  :-)