Scalar Optimisation Part 2

Michael O’Boyle

January 2014
Course Structure

• L1 Introduction and Recap

• 4-5 lectures on classical optimisation
  – 2 lectures on scalar optimisation
  – Last lecture on redundant expressions
  – Today look at dataflow framework and SSA

• 4-5 lectures on high level approaches

• 4-5 lectures on adaptive compilation
Dataflow analysis for redundant expressions: calculate available

$DEExpr(b)$ - subexpressions not overwritten in this block $b$ (local)

$NOTKILLED(b)$ - subexpressions that are not killed (local)

$AVAIL(b) = \bigcap_{p \in pred(b)} (DEExpr(p) \cup (AVAIL(p) \cap NOTKILLED(p)))$

- $DEExpr(b)$ and $NOTKILLED(b)$ can be calculated locally for each basic block $b$

- Initialise $AVAIL(b) = \emptyset$

- Find for each block in turn calculate $AVAIL(b)$ based on predecessors

- Keep repeating the procedure till results stabilise.
AVAIL() set calculation

\[ m = a+b \quad n = a+b \]

\[ p = c + d \quad r = c + d \]

\[ q = a+b \quad r = c+d \]

\[ e = b + 18 \quad s = a + b \quad u = e + f \]

\[ v = a + b \quad w = c + d \quad x = e + f \]

\[ y = a + b \quad z = c + d \]
Calculate Avail(b) for each Basic Block b starting at block A

\[ \text{AVAIL}(A) = \emptyset \]

\[ \text{AVAIL}(B) = (\text{DEExpr}(A) \cup (\text{AVAIL}(A) \cap \text{NOTKILLED}(A))) \]
\[ = \{a + b\} \cup (\emptyset \cap U) = \{a + b\} \]

\[ \text{AVAIL}(G) = (\text{DEExpr}(B) \cup (\text{AVAIL}(B) \cap \text{NOTKILLED}(B))) \]
\[ \cap (\text{DEExpr}(F) \cup (\text{AVAIL}(F) \cap \text{NOTKILLED}(F))) \]
Find available expressions

Post order

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avail1</td>
<td>∅</td>
<td>a+b</td>
<td>a+b</td>
<td>a+b,c+d</td>
<td>a+b,c+d</td>
<td>e+f</td>
<td>c+d</td>
</tr>
<tr>
<td>Avail2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a+b,c+d,e+f</td>
<td>a+b,c+d</td>
</tr>
</tbody>
</table>

Reverse Post order: Finds fixed point on first iteration

<table>
<thead>
<tr>
<th>Node</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avail1</td>
<td>∅</td>
<td>a+b</td>
<td>a+b</td>
<td>a+b,c+d</td>
<td>a+b,c+d</td>
<td>a+b,c+d,e+f</td>
<td>a+b,c+d</td>
</tr>
</tbody>
</table>

Traversal order affects number of iterations to solve equations.

Will solution always terminate?

How many iterations? What class of problems?
Another example: Dataflow analysis for live variables

• A variable \( v \) is live at a point \( p \) if there is a path from \( p \) to a use of \( v \) along which \( v \) is not redefined.

• Useful to eliminate stores of variables no longer needed - useless store elimination

• Useful for detecting uninitialised variables

• Essential for global register allocation

• Determines whether a variable MAY be read after this BB and is therefore a candidate to be put in a register
Equations for live vars

$\text{LiveOut}(b) = \bigcup_{p \in \text{succ}(b)} (\text{UEVar}(p) \cup (\text{LiveOut}(p) \cap \text{NotKilledVar}(p)))$

$\text{UEVar}(p)$ upwardly exposed variables used in $p$ before redefinition

$\text{NotKilledVar}(p)$ var not defined in this block $p$

- Similar to AVAIL

- Depends on successors not predecessors backward vs forward

- AVAIL is an all paths problem ($\cap$) LiveOut any path ($\cup$)

- Can also be solved using iterative algorithm. (How long/terminate?)
Example of LiveOut

\[
\begin{align*}
B0 & \quad i = 1 \\
B1 & \quad a = c = \\
B2 & \quad b = c = d = \\
B3 & \quad a = d = \\
B4 & \quad d = \\
B5 & \quad c = \\
B6 & \quad b = \\
B7 & \quad y = a + b \\
\end{align*}
\]
Solution:

<table>
<thead>
<tr>
<th>UEVar</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVarKill</td>
<td>a,b,c,d,y,z</td>
<td>b,d,i,y,z</td>
<td>a,i,y,z</td>
<td>b,c,i,y,z</td>
<td>a,b,c,i,y,z</td>
<td>a,b,d,i,y,z</td>
<td>a,c,d,i,y,z</td>
<td>a,b,c,d</td>
</tr>
</tbody>
</table>

Reverse Post order

<table>
<thead>
<tr>
<th>Iter</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>a,b,c,d,i</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>a,b,c,d,i</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>a,i</td>
<td>a,b,c,d,i</td>
<td>-</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>a,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>5</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
</tbody>
</table>

5 iterations to fixed point. Is this the quickest solution?
Solution 2

Post order

<table>
<thead>
<tr>
<th>Iter</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>a,c,i</td>
<td>a,b,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,c,d,i</td>
<td>a,b,c,d,i</td>
<td>i</td>
</tr>
</tbody>
</table>

- What is the best order?

- Question: why does all this work?
Semi-lattice

A set $L$ and a meet operator $\wedge$ such that

$\forall a, b, c \in L$

1. $a \wedge a = a$

2. $a \wedge b = b \wedge a$

3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

$\wedge$ imposes an order $a \geq b \rightarrow a \wedge b = b$

Contains a bottom element $\bot$, $\bot \wedge a = \bot$, $a \geq \bot$

Models an ordered finite set of facts
Semi-lattice

• Choose a semi-lattice to represent the facts

• Attach a meaning to each $a \in L$. Each a distinct set of facts

• For each node (basic block) $n$ in the CFG, associate a function $f_n : L \mapsto L$.

• It models the behaviour of the code belonging to $n$

• Avail: Semilattice is $(2^E, \land), E$ the set of all expressions, $\land$ is $\cap$. 

M. O’Boyle  
Scalar Optimisation  
January 2014
Example of LiveOut lattice
Round Robin algorithm

for i = 1 to N
    Avail(b[i]) = 0
change = true
while (change)
    change = false
    for i = 0 to N
        temp = intersect[i] (Def(x) union (Avail x union Nkill(x)))
        if avail(b[i]) != temp
            change = true
            avail(b[i]) = temp
Standard algorithm to solve dataflow. There are faster ones.
Iterative data flow

- If \( f \) is monotone and the semi-lattice bounded then the round robin algorithm terminates and finds a least fixed point

- Given certain technical constraints on \( f \), there is a unique fixed point and order of evaluation does not matter

- Pick an order that converges quickly

- A lot of theory about this. Given certain conditions then a round-robin post-order alg will finish in \( d(G) + 3 \) passes where \( d(G) \) is the loop connectedness

- Most dataflow fits this. Means runs in linear time. Muchnick for more details for more in depth explanation.
Other dataflow analysis

• Reaching definitions: Find all places where a variable was defined and not killed subsequently.

• Very Busy Expressions: An expression is evaluated on all paths leaving a block - used for code hoisting.

• Constant Propagation. Shows that a variable \( v \) has the same value at point \( p \) regardless of control-flow. Allows specialisation.

• Uses a very small lattice and terminates quickly. Easy to express using SSA form.
SSA form

- Most advanced analysis needs to track def and uses of vars rather than basic block summary
- Variables can have multiple definitions and uses
- Need to keep track of which def flows to which use over all possible control-flow paths
- SSA gives a unique name to each definition
- Need $\phi$ nodes to handle merging of control-flow
- Can be constructed in $O(n)$ time. Increasingly standard form.
Example SSA

\[ i = 1 \]
\[ a1 = (a0, a4) \]
\[ a2 = \]
\[ b = \]
\[ c = \]
\[ d = \]
\[ a3 = d = \]
\[ a4 = (a2, a3) \]
\[ y = a4 + b \]
\[ i = i + 1 \]

B0 \rightarrow B1 \rightarrow B2 \rightarrow B4 \rightarrow B7
B1 \rightarrow B3 \rightarrow B5
B2 \rightarrow B4
B4 \rightarrow B5
B5 \rightarrow B6
B6 \rightarrow B7
Algorithms using SSA

• Many dataflow algorithms are considerably simplified using SSA

• Value numbering. Each value has a unique name allowing value numbering on complex control-flow

• $Constants(n) = \bigwedge_{p \in \text{pred}(n)} F_p(\text{Constants}(p))$

• Small lattice $\top > \{-\text{maxint} .. +\text{maxint}\} > \bot$

• Meet operator: $\top \land x = x$, $\bot \land x = \bot$, $c_i \land c_j = c_i$ if $c_i = c_j$ else $\bot$

• $F_p$ depends on the operations in block. Optimistic algorithm
Algorithms using SSA: Constant propagation

Model $F_p$

$x = y$ if $\text{Constants}(p) = \{(x, c_1), (y, c_2), ..\}$ then
$\text{Constants}(p) = \text{Constants}(p) - (x, c_1) \cup (x, c_2)$

- eg update old value of $x$ ($c_1$) with the new value in $y$ ($c_2$)

$x = y \text{ op } z$ if $\text{Constants}(p) = \{(x, c_1), (y, c_2), (z, c_3), ..\}$ then
$\text{Constants}(p) = \text{Constants}(p) - (x, c_1) \cup (x, c_2 \text{ op } c_3)$

- eg update old value of $x$ ($c_1$) with the new value after ($c_2$ op $c_3$)
\[ x_0 = 17 \]
\[ x_1 = \bigcap (x_0, x_2) \]
\[ x_2 = x_1 + i_0 \]

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 ) (when ( i_0 = 0 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>( \top )</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>( 17 \land \top = 17 )</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>( 17 \land 17 = 17 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 ) (when ( i_0 = 1 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>( \top )</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>( 17 \land \top = 17 )</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>( 17 \land 18 = \bot )</td>
</tr>
</tbody>
</table>
Limits and Extensions

- Dataflow assumes that all paths in the CFG are taken hence conservative approximations

- Guarded SSA attempts to overcome this by having additional meet nodes $\gamma, \eta$ and $\mu$ to carry conditional information around


- Array based SSA models access patterns - can be generalised using presburger formula

- Inter-procedural challenging. Pointers destroy analysis! Large research effort in points-to analysis.
Summary

• Levels of optimisations

• Examined dataflow as a generic optimisation framework

• Round robin algorithm and lattices

• Using SSA as a framework for optimisation

• Limits of dataflow - other techniques?

• Next lecture code generation.