Global Register Allocation - 2

Y N Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Principles of Compiler Design
Outline

- Issues in Global Register Allocation (in part 1)
- The Problem (in part 1)
- Register Allocation based in Usage Counts
- Linear Scan Register allocation
- Chaitin’s graph colouring based algorithm
The Problem

- Global Register Allocation assumes that allocation is done beyond basic blocks and usually at function level.
- Decision problem related to register allocation:
  - Given an intermediate language program represented as a control flow graph and a number $k$, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads or stores are introduced, and at most $k$ registers are used.
- This problem has been shown to be NP-hard (Sethi 1970).
- Graph colouring is the most popular heuristic used.
- However, there are simpler algorithms as well.
Conflicting variables

- Two variables interfere or conflict if their live ranges intersect
  - A variable is live at a point \( p \) in the flow graph, if there is a use of that variable in the path from \( p \) to the end of the flow graph
  - The live range of a variable is the smallest set of program points at which it is live.
  - Typically, instruction no. in the basic block along with the basic block no. is the representation for a point.
Example

If (cond)          A not live
    then A =          
    else B =
X: if (cond)         B not live
    then = A
    else = B

A and B both live

Live range of A: B2, B4 B5
Live range of B: B3, B4, B6
Global Register Allocation via Usage Counts (for Single Loops)

- Allocate registers for variables used within loops
- Requires information about liveness of variables at the entry and exit of each basic block (BB) of a loop
- Once a variable is computed into a register, it stays in that register until the end of the BB (subject to existence of next-uses)
- Load/Store instructions cost 2 units (because they occupy two words)
Global Register Allocation via Usage Counts (for Single Loops)

1. For every usage of a variable $v$ in a BB, until it is first defined, do:
   - $\text{savings}(v) = \text{savings}(v) + 1$
   - after $v$ is defined, it stays in the register any way, and all further references are to that register

2. For every variable $v$ computed in a BB, if it is live on exit from the BB,
   - count a savings of 2, since it is not necessary to store it at the end of the BB
Global Register Allocation via Usage Counts (for Single Loops)

- Total savings per variable $v$ are

$$\sum_{B \in \text{Loop}} (\text{savings}(v, B) + 2 \times \text{liveandcomputed}(v, B))$$

- $\text{liveandcomputed}(v, B)$ in the second term is 1 or 0

- On entry to (exit from) the loop, we load (store) a variable live on entry (exit), and lose 2 units for each

- But, these are “one time” costs and are neglected

- Variables, whose savings are the highest will reside in registers
Global Register Allocation via Usage Counts (for Single Loops)

Savings for the variables

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(0+2)+(1+0)+(1+0)+(0+0) = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>(3+0)+(0+0)+(0+0)+(0+2) = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>(1+0)+(1+0)+(0+0)+(1+0) = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>(0+2)+(1+0)+(0+0)+(1+0) = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>(0+2)+(0+0)+(1+0)+(0+0) = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>(1+0)+(1+0)+(0+2)+(0+0) = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If there are 3 registers, they will be allocated to the variables, a, b, and d

\[ a = b \times c \]
\[ d = b - a \]
\[ e = b / f \]

b
\[ e = a - f \]
\[ e = d + c \]
\[ f = e \times a \]
\[ b = c - d \]
Global Register Allocation via Usage Counts (for Nested Loops)

- We first assign registers for inner loops and then consider outer loops. Let \( L_1 \) nest \( L_2 \)

- For variables assigned registers in \( L_2 \), but not in \( L_1 \)
  - load these variables on entry to \( L_2 \) and store them on exit from \( L_2 \)

- For variables assigned registers in \( L_1 \), but not in \( L_2 \)
  - store these variables on entry to \( L_2 \) and load them on exit from \( L_2 \)

- All costs are calculated keeping the above rules
Global Register Allocation via Usage Counts (for Nested Loops)

- **case 1:** variables $x, y, z$ assigned registers in L2, but not in L1
  - Load $x, y, z$ on entry to L2
  - Store $x, y, z$ on exit from L2

- **case 2:** variables $a, b, c$ assigned registers in L1, but not in L2
  - Store $a, b, c$ on entry to L2
  - Load $a, b, c$ on exit from L2

- **case 3:** variables $p, q$ assigned registers in both L1 and L2
  - No special action
A Fast Register Allocation Scheme

- Linear scan register allocation (Poletto and Sarkar 1999) uses the notion of a live interval rather than a live range.
- Is relevant for applications where compile time is important, such as in dynamic compilation and in just-in-time compilers.
- Other register allocation schemes based on graph colouring are slow and are not suitable for JIT and dynamic compilers.
Linear Scan Register Allocation

- Assume that there is some numbering of the instructions in the intermediate form.
- An interval $[i,j]$ is a live interval for variable $v$ if there is no instruction with number $j' > j$ such that $v$ is live at $j'$ and no instruction with number $i' < i$ such that $v$ is live at $i$.
- This is a conservative approximation of live ranges: there may be subranges of $[i,j]$ in which $v$ is not live but these are ignored.
Live Interval Example

v live

sequentially numbered instructions

v live

i': ...

i: ...

j: ...

j': ...

v live

i' does not exist

i – j: live interval for variable v

j' does not exist
Example

If (cond)
then A=
else B=

X: if (cond)
then =A
else = B

A NOT LIVE HERE

LIVE INTERVAL FOR A

If (cond)
A=

If (cond)
B=

If (cond)
= A

= B

T

F
Live Intervals

- Given an order for pseudo-instructions and live variable information, live intervals can be computed easily with one pass through the intermediate representation.
- Interference among live intervals is assumed if they overlap.
- Number of overlapping intervals changes only at start and end points of an interval.
The Data Structures

- Live intervals are stored in the sorted order of increasing start point.
- At each point of the program, the algorithm maintains a list (active list) of live intervals that overlap the current point and that have been placed in registers.
- active list is kept in the sorted order of increasing end point.
Example

Active lists (in order of increasing end pt)

Active(A) = {i1}
Active(B) = {i1, i5}
Active(C) = {i8, i5}
Active(D) = {i7, i4, i11}

Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers are enough for computation without spills
The Algorithm (1)

\[
\begin{align*}
\{ \text{active} & := [ ]; \\
\text{for each live interval } i, \text{ in order of increasing start point } & \text{ do} \\
\{ \text{ExpireOldIntervals}(i); \\
\text{if length(active) } &= \text{ R then SpillAtInterval}(i); \\
\text{else } &\{ \text{register}[i] := \text{a register removed from the pool of free registers;} \\
\text{add } i \text{ to active, sorted by increasing end point} \\
\} \\
\} \\
\}
\end{align*}
\]
The Algorithm (2)

ExpireOldIntervals (i)
{
    for each interval j in active, in order of increasing end point do
        if endpoint[j] > startpoint[i] then continue
        else
            remove j from active;
            add register[j] to pool of free registers;
    }
}
The Algorithm (3)

SpillAtInterval (i)
{ spill := last interval in active; /* last ending interval */
  if endpoint [spill] ≥ endpoint [i] then
    { register [i] := register [spill];
      location [spill] := new stack location;
      remove spill from active;
      add i to active, sorted by increasing end point;
    }
  else location [i] := new stack location;
}
Active lists (in order of increasing end pt)

Active(A) = \{i1\}
Active(B) = \{i1, i5\}
Active(C) = \{i8, i5\}
Active(D) = \{i7, i4, i11\}

Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers are enough for computation without spills
Example 2

1,2: give A,B register
3: Spill C since endpoint[C] > endpoint [B]
4: A expires, give D register
5: B expires, E gets register

2 registers available
Example 3

1,2 : give A,B register
3: Spill B since endpoint[B] > endpoint [C]
give register to C

4: A expires, give D register
5: C expires, E gets register
Complexity of the Linear Scan Algorithm

- If $V$ is the number of live intervals and $R$ the number of available physical registers, then if a balanced binary tree is used for storing the active intervals, complexity is $O(V \log R)$.
  - Active list can be at most ‘$R$’ long
  - Insertion and deletion are the important operations

- Empirical results reported in literature indicate that linear scan is significantly faster than graph colouring algorithms and code emitted is at most 10% slower than that generated by an aggressive graph colouring algorithm.
Chaitin’s Formulation of the Register Allocation Problem

- A graph colouring formulation on the interference graph
- Nodes in the graph represent either live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists are used to represent the graph.
Chaitin’s Formulation of the Register Allocation Problem

- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour
  - The number of colours available is the number of registers available on the machine
  - A k-colouring of the interference graph is mapped onto an allocation with k registers
Example

- Two colourable
- Three colourable
Idea behind Chaitin’s Algorithm

- Choose an arbitrary node of degree less than $k$ and put it on the stack
- Remove that vertex and all its edges from the graph
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than $k$
- At some point, if all vertices have degree greater than or equal to $k$, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in a colour not used by neighbours (reuse colours as far as possible)