Global Register Allocation - Part 1

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

NPTEL Course on Compiler Design

Outline

- Issues in Global Register Allocation
- The Problem
- Register Allocation based in Usage Counts
- Linear Scan Register allocation
- Chaitin's graph colouring based algorithm

Some Issues in Register Allocation

- Which values in a program reside in registers? (register allocation)
- In which register? (register assignment)
  - The two together are usually loosely referred to as register allocation
- What is the unit at the level of which register allocation is done?
  - Typical units are basic blocks, functions and regions.
  - RA within basic blocks is called local RA
  - The other two are known as global RA
  - Global RA requires much more time than local RA

Some Issues in Register Allocation

- Phase ordering between register allocation and instruction scheduling
  - Performing RA first restricts movement of code during scheduling – not recommended
  - Scheduling instructions first cannot handle spill code introduced during RA
    - Requires another pass of scheduling
- Tradeoff between speed and quality of allocation
  - In some cases e.g., in Just-In-Time compilation, cannot afford to spend too much time in register allocation.

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

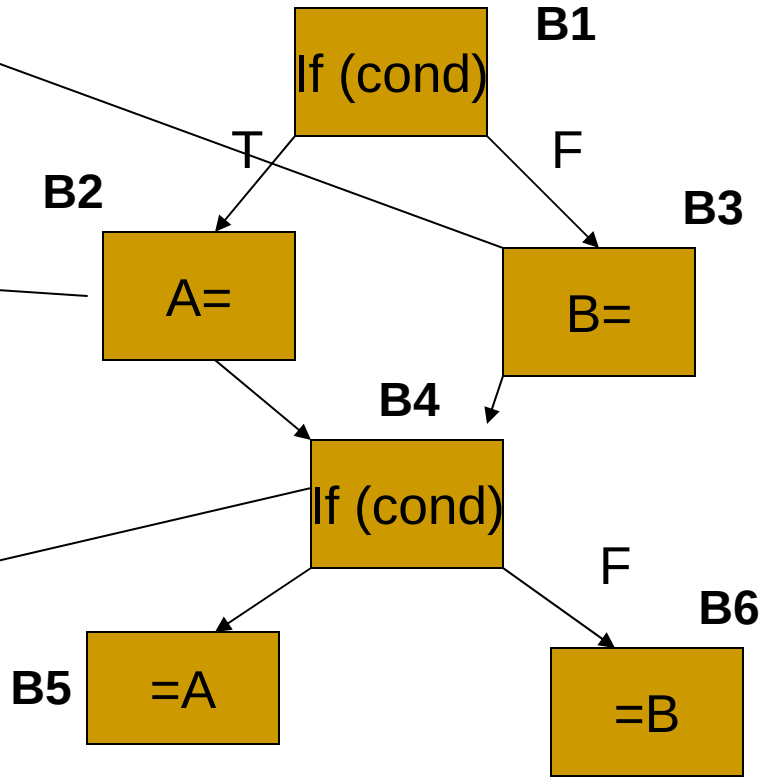
Conflicting variables

- 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
  - A live range of a variable is the set of program points (in the flow graph) 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

Live range of A: B2, B4 B5
Live range of B: B3, B4, B6

```
   If (cond)          A not live
       then A =
       else B =
X: if (cond)          B not live
        then = A
        else = B
-----------------------------
   A and B both live
```



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 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:
   - savings(v) = 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

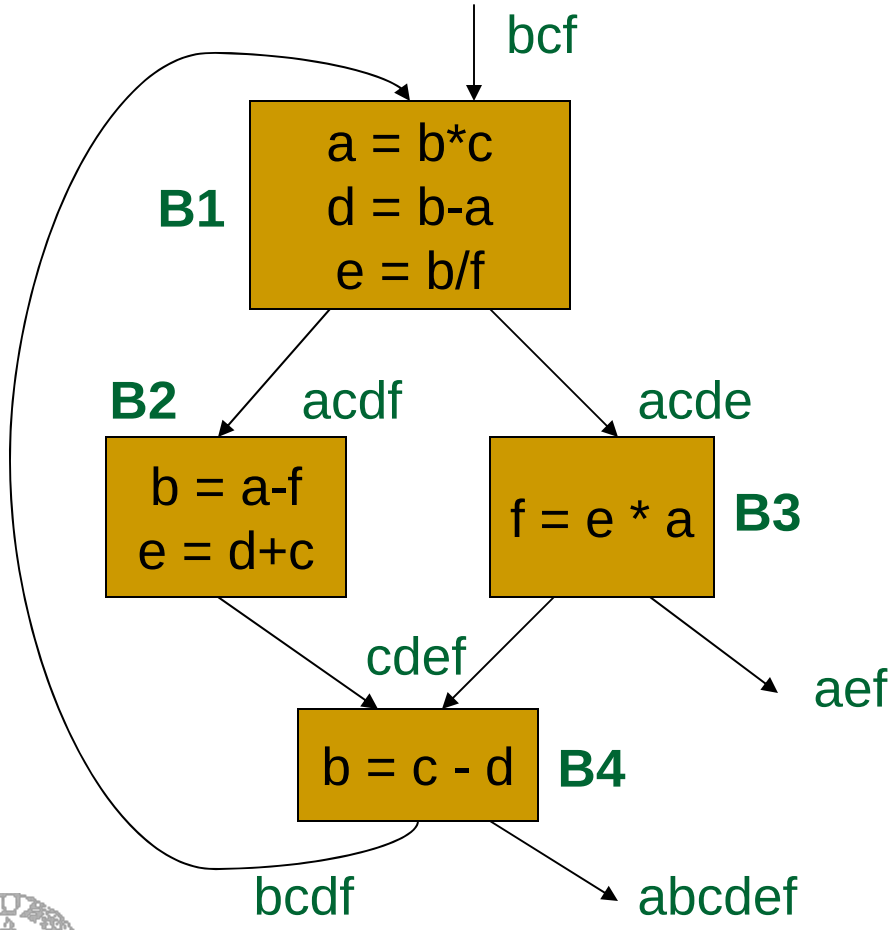
Global Register Allocation via Usage Counts (for Single Loops)

- Total savings per variable ***v*** are

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

  - *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

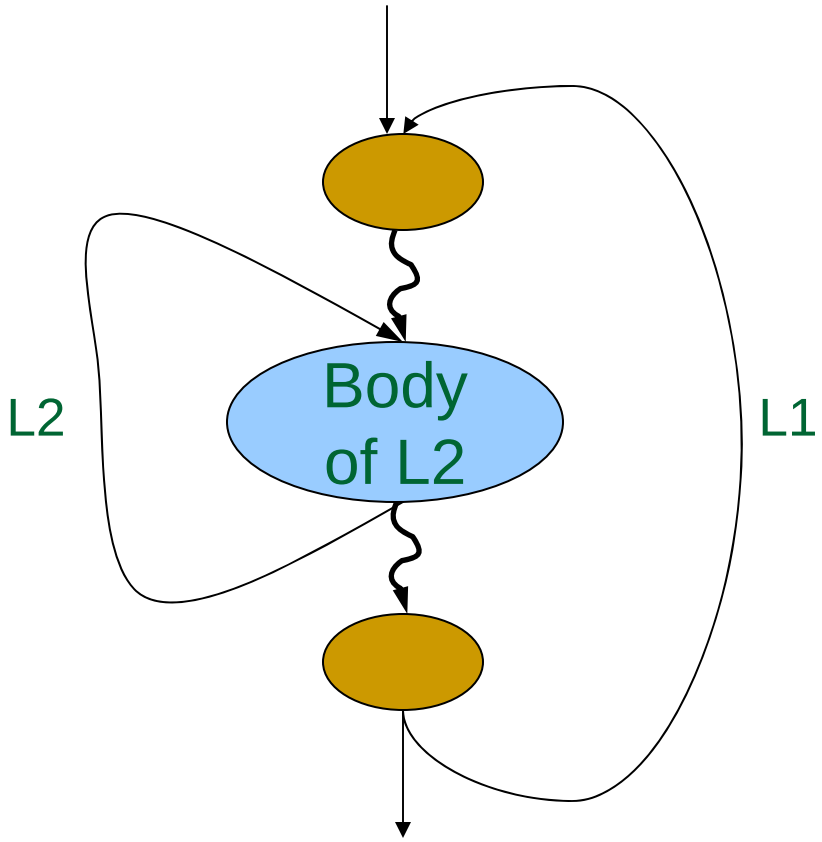
	B1	B2	B3	B4
a:	(0+2)	+(1+0)	+(1+0)	+(0+0) = 4
b:	(3+0)	+(0+0)	+(0+0)	+(0+2) = 5
c:	(1+0)	+(1+0)	+(0+0)	+(1+0) = 3
d:	(0+2)	+(1+0)	+(0+0)	+(1+0) = 4
e:	(0+2)	+(0+2)	+(1+0)	+(0+0) = 5
f:	(1+0)	+(1+0)	+(0+2)	+(0+0) = 4

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

Global Register Allocation via Usage Counts (for Nested Loops)

- We first assign registers for inner loops and then consider outer loops. Let L1 nest L2
- For variables assigned registers in L2, but not in L1
  - load these variables on entry to L2 and store them on exit from L2
- For variables assigned registers in L1, but not in L2
  - store these variables on entry to L2 and load them on exit from L2
- 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

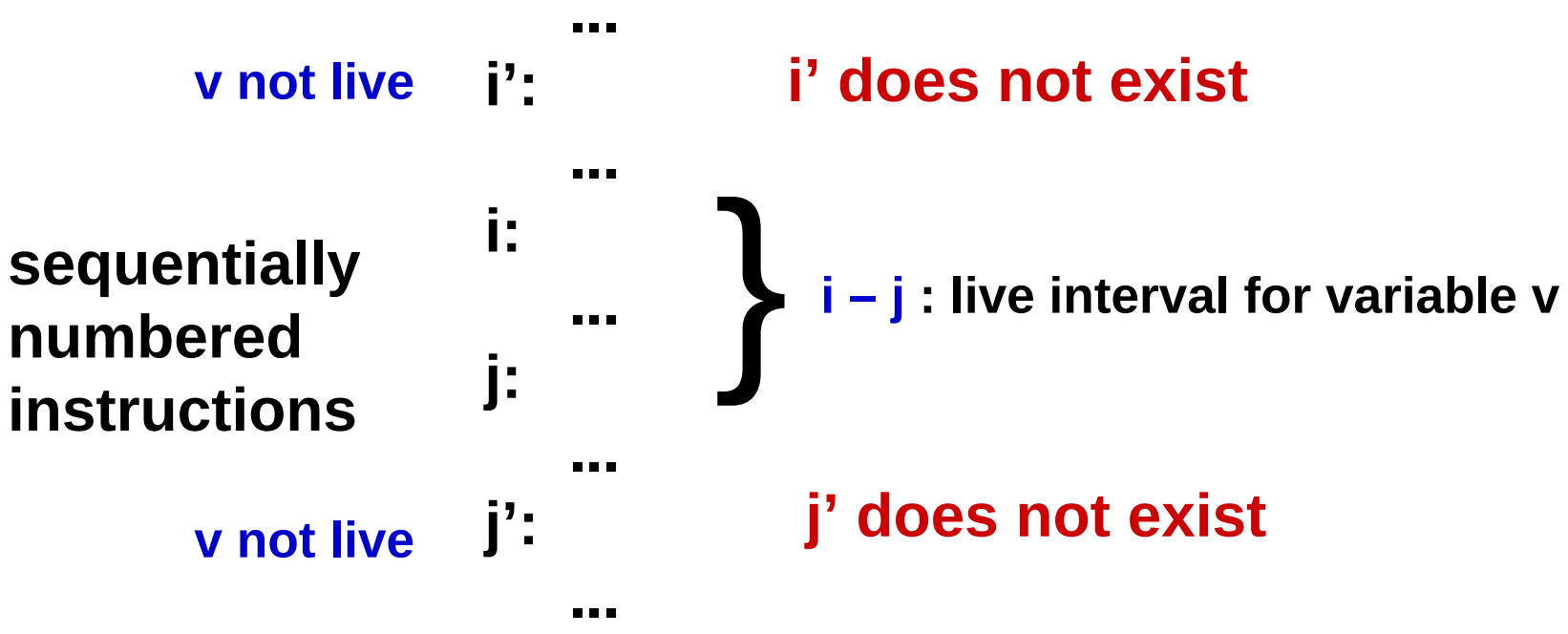
A Fast Register Allocation Scheme

- 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 raph 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

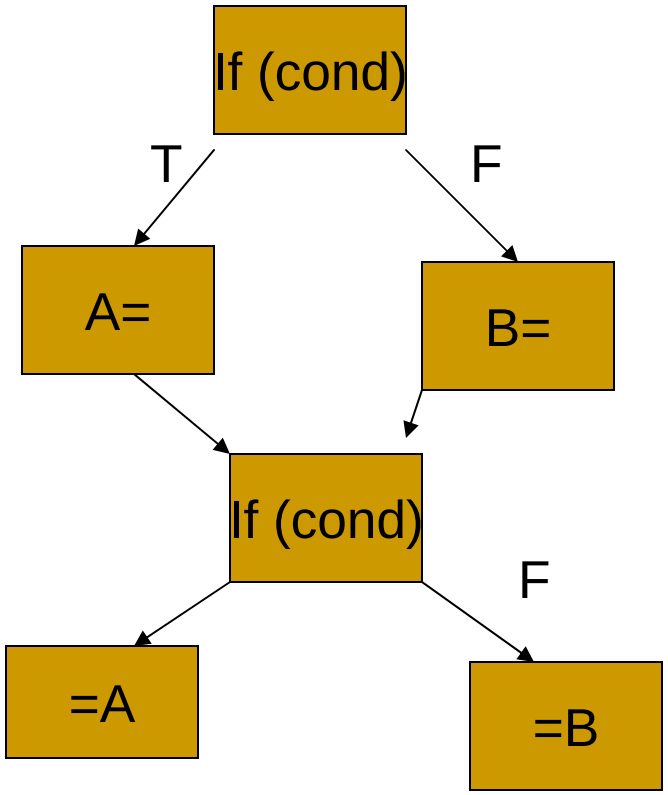


Example

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

A NOT LIVE HERE

LIVE INTERVAL FOR A



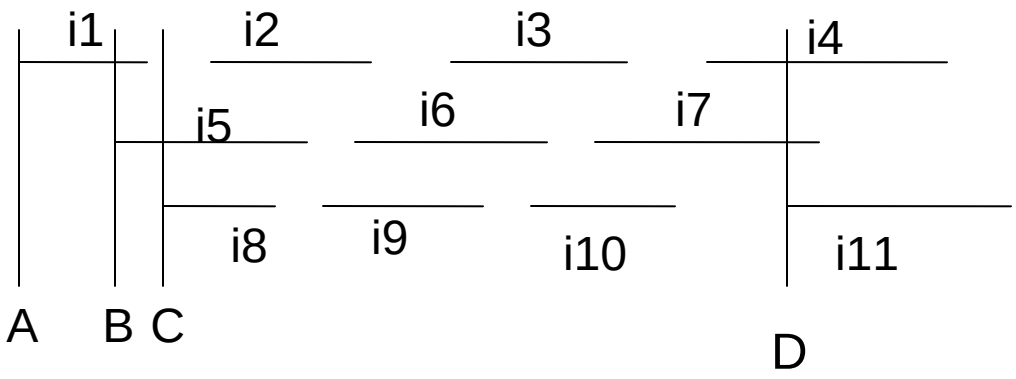
Live Intervals

- 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 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 enough for computation without spills