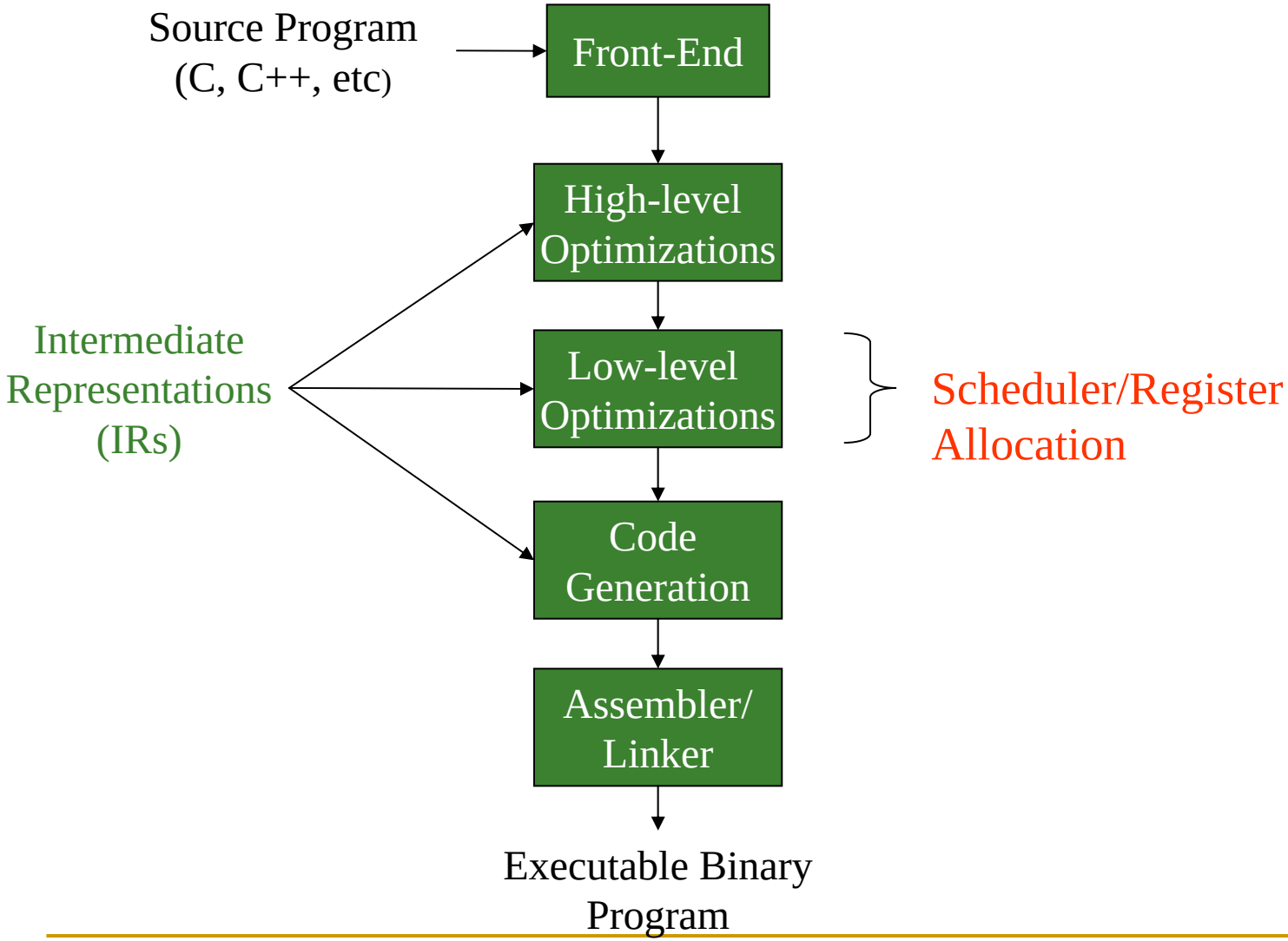
Topic 6
Register Allocation

Recap: Structure of an Optimizing Compiler



Rationale for Separating Register Allocation from Scheduling

- Each of Scheduling and Register Allocation are hard to solve individually, let alone solve *globally* as a combined optimization.

- So, solve each optimization *locally* and heuristically "patch up" the two stages.

Why Register Allocation?

- Storing and accessing variables from registers is much faster than accessing data from memory.

  The way operations are performed in load/store (RISC) processors.

- Therefore, in the interests of performance—if not by necessity—variables ought to be stored in registers.
- For performance reasons, it is useful to store variables as long as possible, once they are loaded into registers.
- Registers are bounded in number (say 32.)
- Therefore, “register-sharing” is needed over time.

The Goal

- *Primarily* to assign registers to variables.
- However, the allocator runs out of registers quite often.
- Decide which variables to "flush" out of registers to free them up, so that other variables can be bought in.

  This important indirect consequence of allocation is referred to as *spilling.*

Register Allocation and Assignment

Allocation: identifying program values (virtual registers, live ranges) and program points at which values should be stored in a physical register.

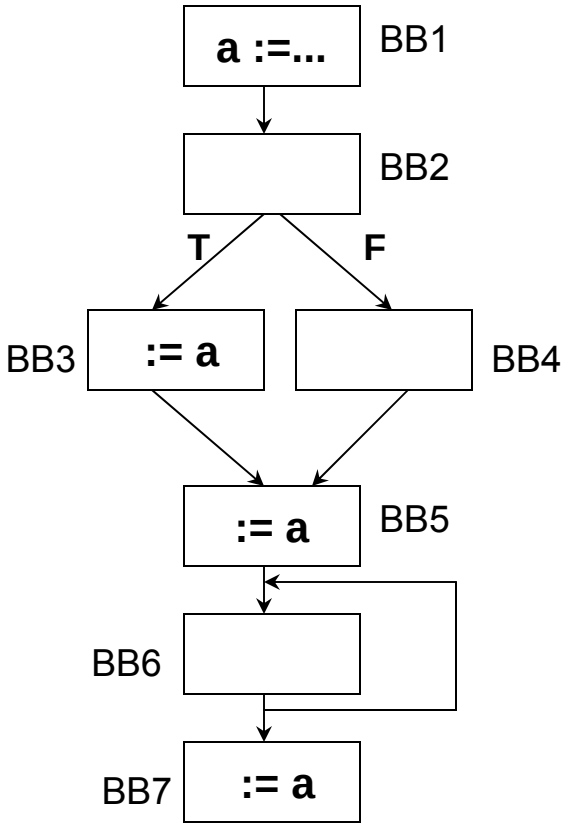
Program values that are not allocated to registers are said to be *spilled*.

Assignment: identifying which physical register should hold an allocated value at each program point.

Live Ranges

Live range of virtual register *a* = (BB1, BB2, BB3, BB4, BB5, BB6, BB7).

Def-Use chain of virtual register *a* = (BB1, BB3, BB5, BB7).



Computing Live Ranges

Using data flow analysis, we compute for each basic block:

- In the forward direction, the *reaching* attribute.

  A variable is reaching block i if a definition or use of the variable reaches the basic block along the edges of the CFG.

- In the backward direction, the *liveness* attribute.

  A variable is live at block i if there is a direct reference to the variable at block i or at some block j that succeeds i in the CFG, provided the variable in question is *not redefined* in the interval between i and j.

Computing Live Ranges (Contd.)

The live range of a variable is the intersection of basic-blocks in CFG nodes in which the variable is live, and the set which it can reach.

Local Register Allocation

- Perform register allocation one basic block (or super-block or hyper-block) at a time
- Must note virtual registers that are live on entry and live on exit of a basic block - later perform reconciliation
- During allocation, spill out all live outs and bring all live ins from memory
- Advantage: very fast

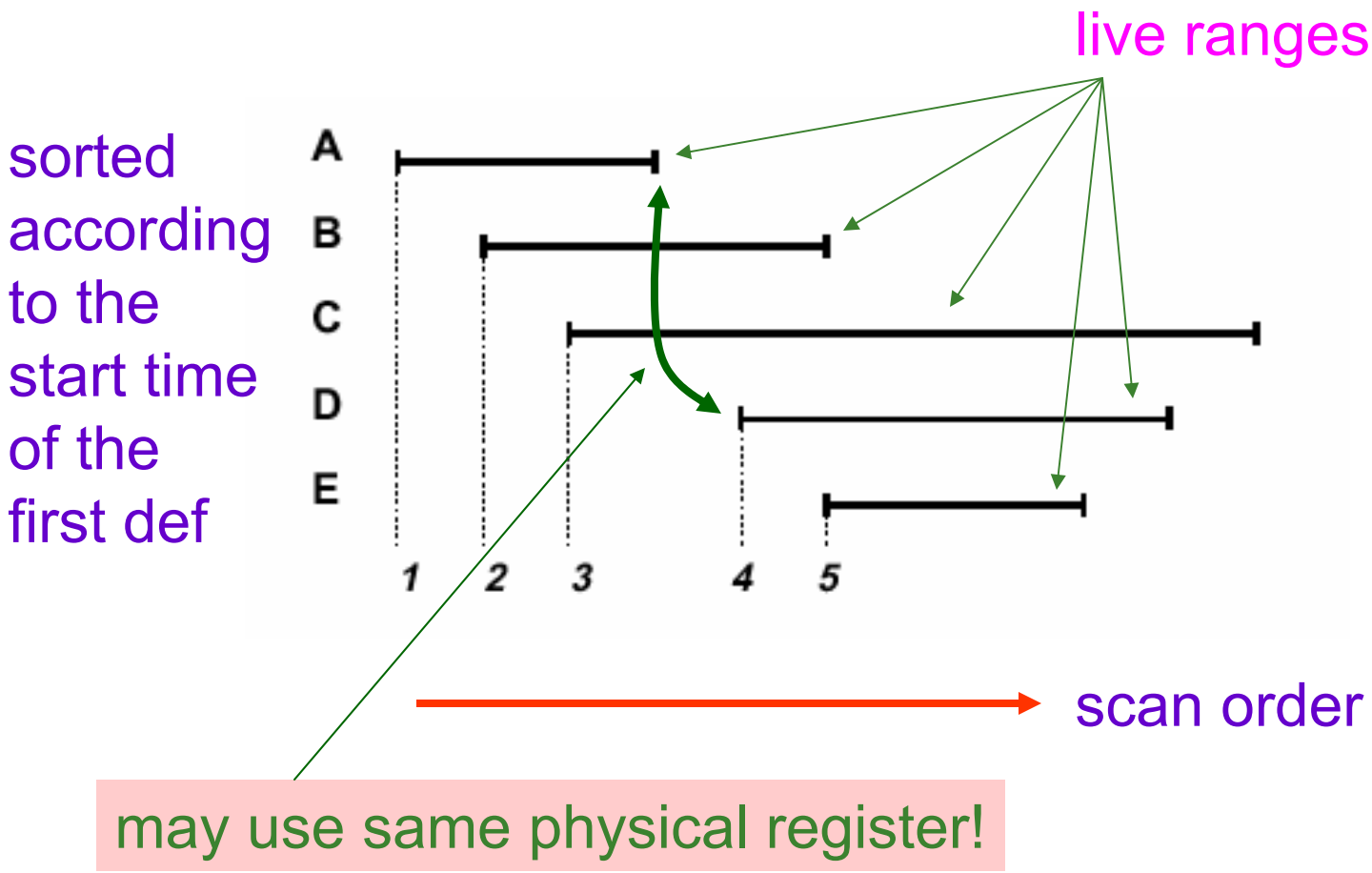
Local Register Allocation - Reconciliation Codes

- After completing allocation for all blocks, we need to reconcile differences in allocation
- If there are spare registers between two basic blocks, replace the spill out and spill in code with register moves

Linear Scan Register Allocation

- A simple local allocation algorithm
- Assume code is already scheduled
- Build a linear ordering of live ranges (also called live intervals or lifetimes)
- Scan the live range and assign registers until you run out of them - then spill

Linear Scan RA



The Linear Scan Algo

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    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
```

actual spill

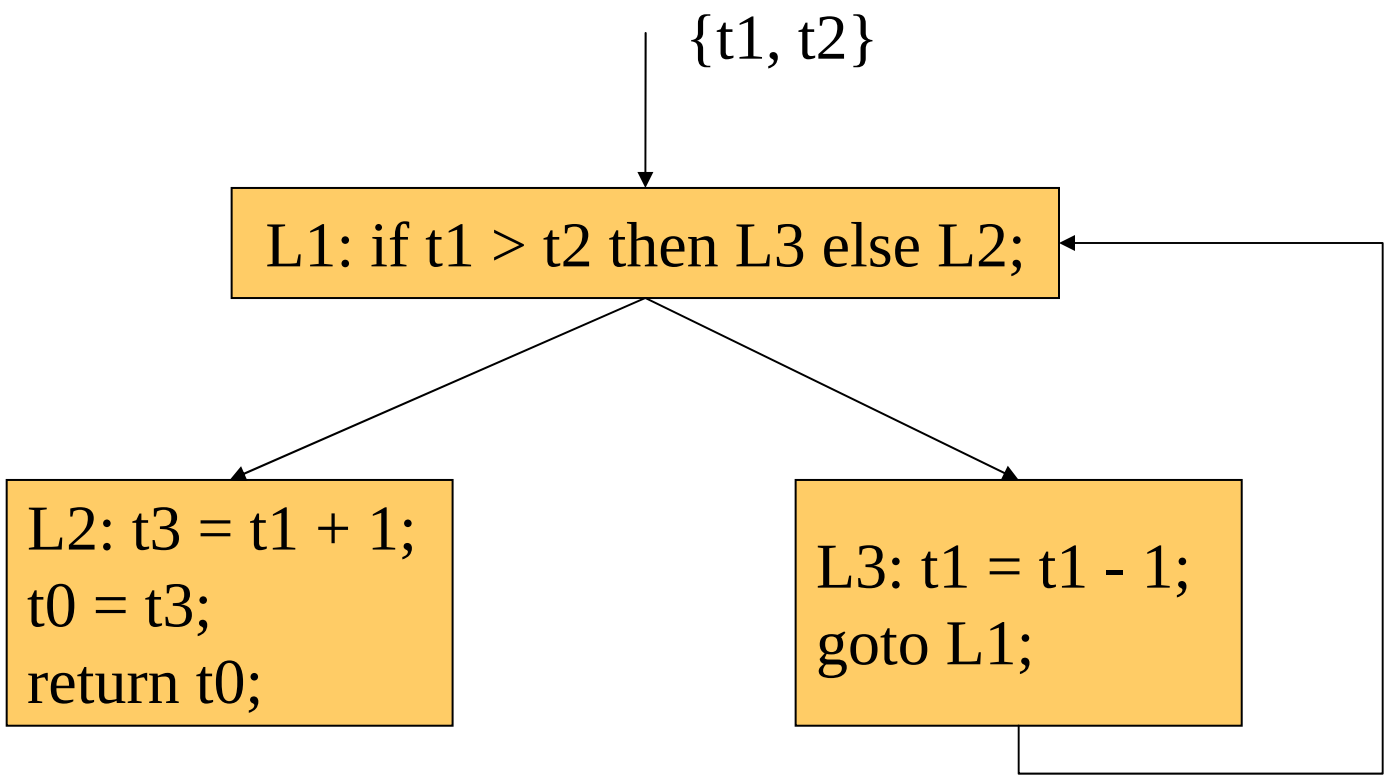
Source:
M. Poletto & V. Sarkar., "Linear Scan Register Allocation", ACM TOPLAS, Sep 1999.

Summary of the Linear Scan Register Allocation(LSR)

- LSR can be seen as 4 simple steps
  - Order the instructions in linear fashion
    - Many have proposed heuristics for finding the best linear order
  - Calculate the set of live intervals
    - Each temporary is given a live interval
  - Allocate a register to each interval
    - If a register is available then allocation is possible
    - If a register is not available then an already allocated register is chosen (register spill occurs)
  - Rewrite the code according to the allocation
    - Actual registers replace temporary or virtual registers
    - Spill code is generated

Example: Applying the LSR Algorithm

- Given the following CFG



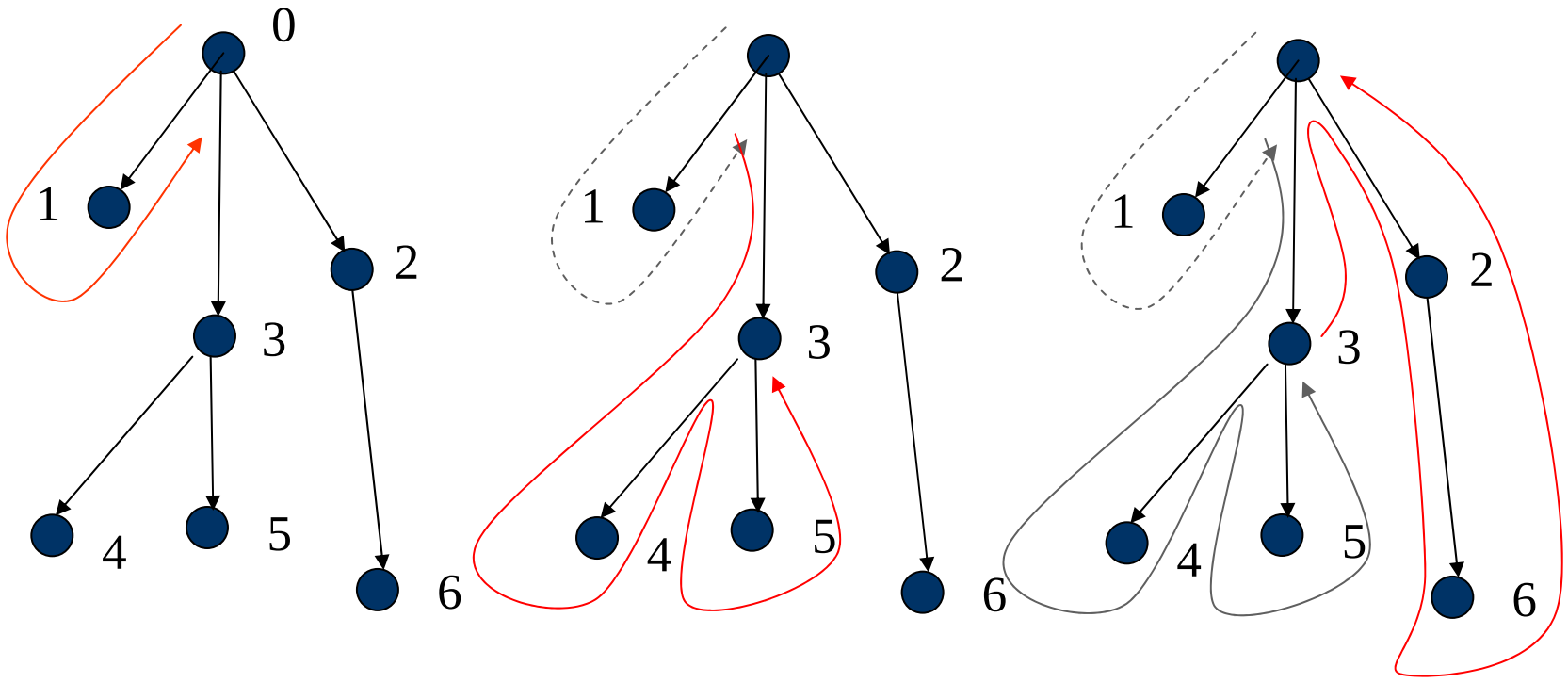
Example: Step 1 of LSR

- Find a linear ordering of the instructions
  - The a priori choice of ordering affects performance
  - Exhaustive search for optimal order is not feasible
  - Poletto and Sarkar
    - Compare depth-first ordering with ordering found in the IR

```
0: L1: if t1 > t2 then L3 else L2;
1: L3: t1 = t1 – 1;
2:       goto L1;
3: L2: t3 = t1 + 1;
4:       t0 = t3;
5:       return t0;
```

The IR ordering

Depth-First Ordering



- Result: 0, 1, 3, 4, 5, 2, 6

Example: The DF Ordering and the IR Ordering

- Applying DF ordering to our CFG

```
0: L1: if t1 > t2 then L3 else L2;
1: L2: t3 = t1 + 1;
2:      t0 = t3;
3:      return t0;
4: L3: t1 = t1 – 1;
5:      goto L1;
```

DF Ordering

```
0: L1: if t1 > t2 then L3 else L2;
1: L3: t1 = t1 – 1;
2:      goto L1;
3: L2: t3 = t1 + 1;
4:      t0 = t3;
5:      return t0;
```

IR Ordering

Example: Step 2, Compute Live intervals for DF ordering

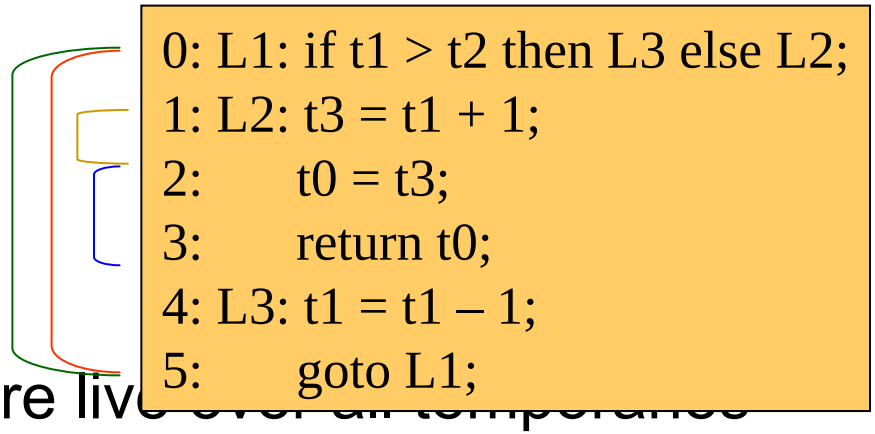
- There are 4 temporaries or virtual registers
  - t0, t1, t2, t3
- Live intervals for DF Ordering

  t0[2,3]

  t1[0,5]

  t2[0,5]

  t3[1,2]

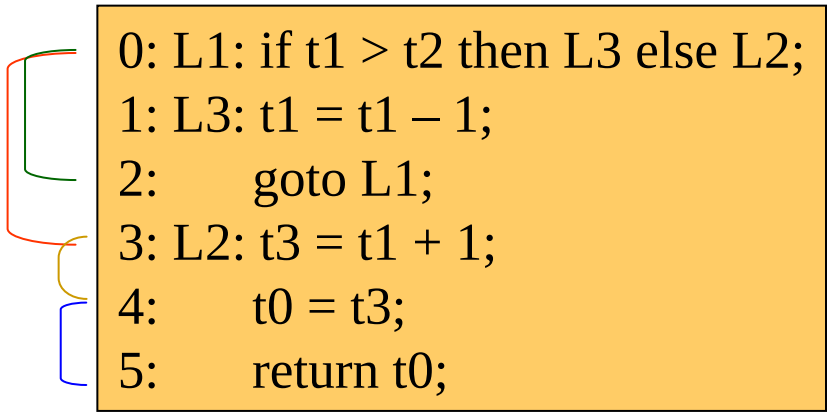


- Shows the t2 and t1 are live over all temporaries

Example: Step 2, Compute Live intervals for IR ordering

- Can you compute the live intervals?
- Live intervals for IR Ordering

t0[4,5]
t1[0,3]
t2[0,2]
t3[3,4]



Shows the only overlap is t2 and t1 with each other

Example: Step 3, Allocate Registers to Intervals

- 3 lists are maintained during this process
  - *Free* : set of available registers
  - *Alloc* : set of allocated registers
  - *Active*: list of active intervals ordered by increasing end points
- Registers are assigned in the following manner
  - Order the intervals in order of increasing start point
  - Scan the list of intervals, select the next t_i
    - Free all registers assigned to intervals in *Active* whose interval is less than or equal to the start of t_i
    - If a free register exists in *Free*, then allocate it
    - If *Free* is empty, then spill in the following way
      - If last interval on the *Active* list ends beyond the interval for t_i, then t_i is given that register, and t_i's interval is added to *Active*
      - If t_i's interval ends at the same point or beyond the last interval in *Active*, then t_i is given a stack location.

Example: Applying Step 3 to the DF ordering

Given 2 regs

List of live intervals

t1[0,5]
t2[0,5]
t3[1,2]
t0[2,3]

- Free ={r1, r2},
  Active = {},
  Alloc = {}
  - Looking at t1, Allocate r1
- Free ={r2},
  Active = {t1:[0,5]},
  Alloc = {r1:t1}
  - Looking at t2, Allocate r2
- Alloc = {r2:t2, r1:t1}
- Active = {t2:[0,5],t1:[0,5]}
- Now Free = {} and there are still more intervals to process…

Start of interval = 0

Start of interval = 0

Example: Applying Step 3 to the DF ordering

Given 2 regs

List of live intervals

t1[0,5]
t2[0,5]
t3[1,2]
t0[2,3]

- Free = {},

  Active = {t2:[0,5], t1:[0,5]},

  Alloc = {r2:t2, r1:t1}

  - Looking at t3
  - Free is empty (spill needed)
  - End of t1 > end of t3
  - t3 is allocated r1, t1 on stack

Start of interval = 1

- Free = {},

  Active = {t3:[1,2], t2:[0,5]} Alloc = {r1:t3, r2:t2}

  - Looking at t0,
  - Free is empty (spill needed)
  - End of t2 > end of t0
  - t0 is allocated r2, t2 on stack

Start of interval = 2

Example: Step 4, Rewriting the code

- Code is rewritten with assigned registers and spill code inserted
- Spill code is additional code that may increase cycle time

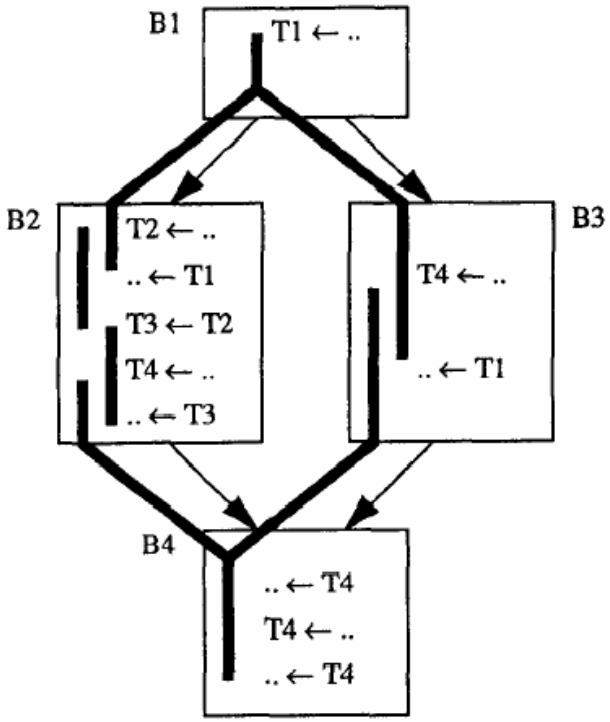
```
0: L1: if t1(r1) > t2(r2) then L3 else L2;
1:    spill t1(r1)  and reassign r1(cond #1)
2: L2: t3(r1) = t1(stk) + 1;
3:    spill t0(r2) and reassign r2(cond  #1)
4:       t0(r2) = t3(r1);
5:       return t0(r2);
6: L3: t1(stk) = t1(stk) – 1;
7:       goto L1;
```

- What is the minimum number of registers needed to guarantee no spills?

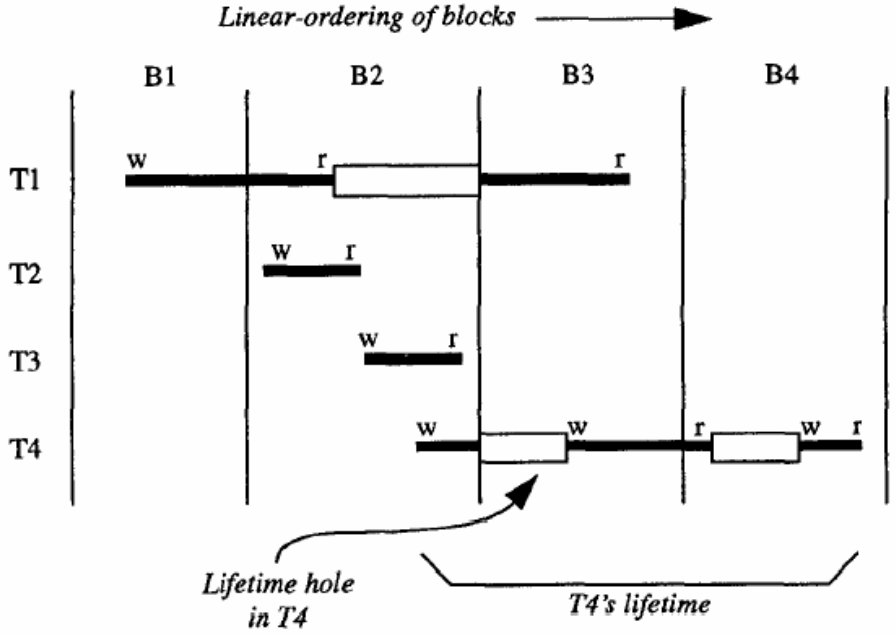
Second Chance Linear Scan

- O. Traub, G. Holloway, and M.D. Smith, “Quality and Speed in Linear-Scan Register Allocation”, SIGPLAN ‘98
- Considers “holes” in live ranges
- Considers register allocation as a bin-packing problem
- Performs register allocation and spill code generation at the same time

Holes in Live Ranges



(a) An example CFG with temporary lifetimes overlaid.



(b) A linear ordering for the example CFG with lifetime holes indicated for each temporary.

Bin-packing

- The binpacking problem: determine how to put the most objects in the least number of fixed space bins
- More formally, find a partition and assignment of a set of objects such that constraint is satisfied or an objective function is minimized (or maximized)
- In register allocation, the constraint is that overlapping live ranges cannot be assigned to the same bin (register)
- Another way of looking at linear scan

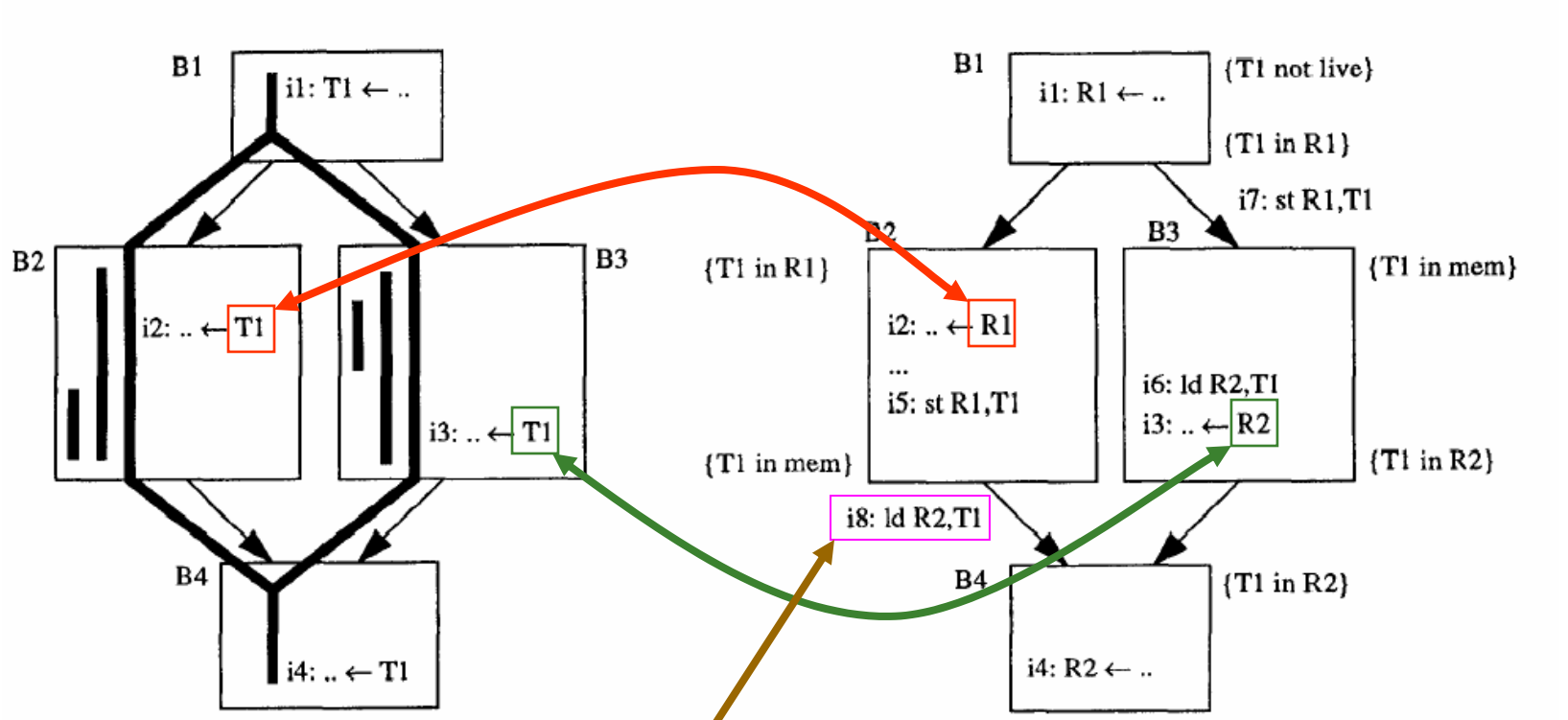
Working with holes

- We can allocate two non-overlapping live ranges to the same physical register
- We can assign two live ranges to the same physical register if one fits entirely into the hole of another

Second Chance Linear Scan

- Suppose we encounter variable *t*, and we assigned a register to it by spilling out variable *u* currently occupying that register
- When *u* is needed again, it may be loaded into a different register (it gets a "second chance")
- It will stay till its lifetime ends or it is evicted again
- Problem: Can cause inconsistent register allocation across the same live range
  - Solution: resolution code has to be inserted

Insertion of Resolution Code



(a) An example CFG before allocation. The CFG contains 5 temporary lifetimes, but only T1's references are shown.

(b) The CFG after allocation. Only instructions associated with T1 are shown. The linear allocation order is B1-B2-B3-B4. The allocation assumptions for T1 before resolution are shown as sets at the top and bottom of each block.

resolution code

Performance

Benchmark	Instruction counts			Run time (sec)		
	Second-chance binpacking	Graph coloring	Ratio (binpack/GC)	Second-chance binpacking	Graph coloring	Ratio (binpack/GC)
alvinn	5859032035	5850062031	1.002	20.56	20.67	0.995
doduc	1610607538	1565260889	1.029	7.36	7.23	1.018
eqntott	2782873030	2777476231	1.002	6.92	6.90	1.003
espresso	1510435454	1390526882	1.086	3.54	3.34	1.060
fpppp	6775315066	6262634084	1.082	25.79	24.73	1.043
li	9878244999	9694580392	1.019	23.91	24.76	0.966
tomcatv	6531688057	6531662363	1.000	14.29	14.36	0.995
compress	94956007702	91999060755	1.032	281.30	275.79	1.020
m88ksim	1112471957	1101374080	1.010	2.97	2.90	1.024
sort	1030126044	989670114	1.041	4.35	4.02	1.082
wc	1046734	1046722	1.000	0.92	0.91	1.011

Table 1: A comparison of the dynamic instruction counts and the run times of executables using either our second-chance binpacking approach or George/Appel's graph-coloring approach.

Appendix A

Global Register Allocation

Global Register Allocation

- Local register allocation does not store data in registers across basic blocks.

  Local allocation has poor register utilization $\Rightarrow$ global register allocation is essential.

- Simple global register allocation: allocate most “active” values in each inner loop.
- Full global register allocation: identify live ranges in control flow graph, allocate live ranges, and split ranges as needed.

  Goal: select allocation so as to minimize number of load/store instructions performed by optimized program.

Topological Sorting

- Given a directed graph, $G = \langle V, E \rangle$, we can define a topological ordering of the nodes

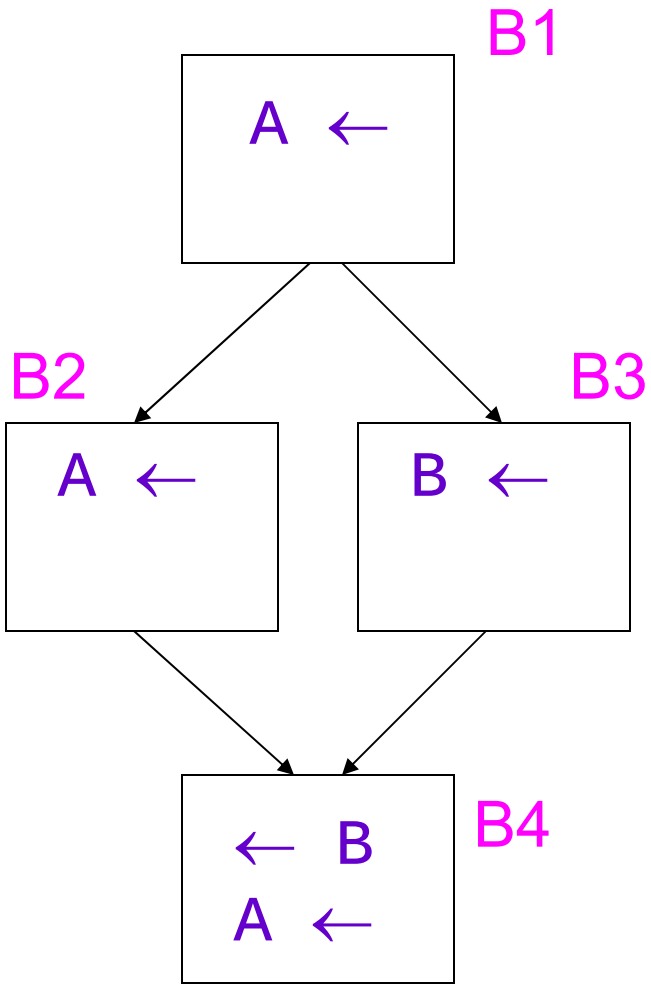
- Let $T = \{v_1, v_2, ..., v_n\}$ be an enumeration of the nodes of V, T is a topological ordering if $v_i \rightarrow v_j \in E$, then $i < j$ (i.e. v_i comes before v_j in T)

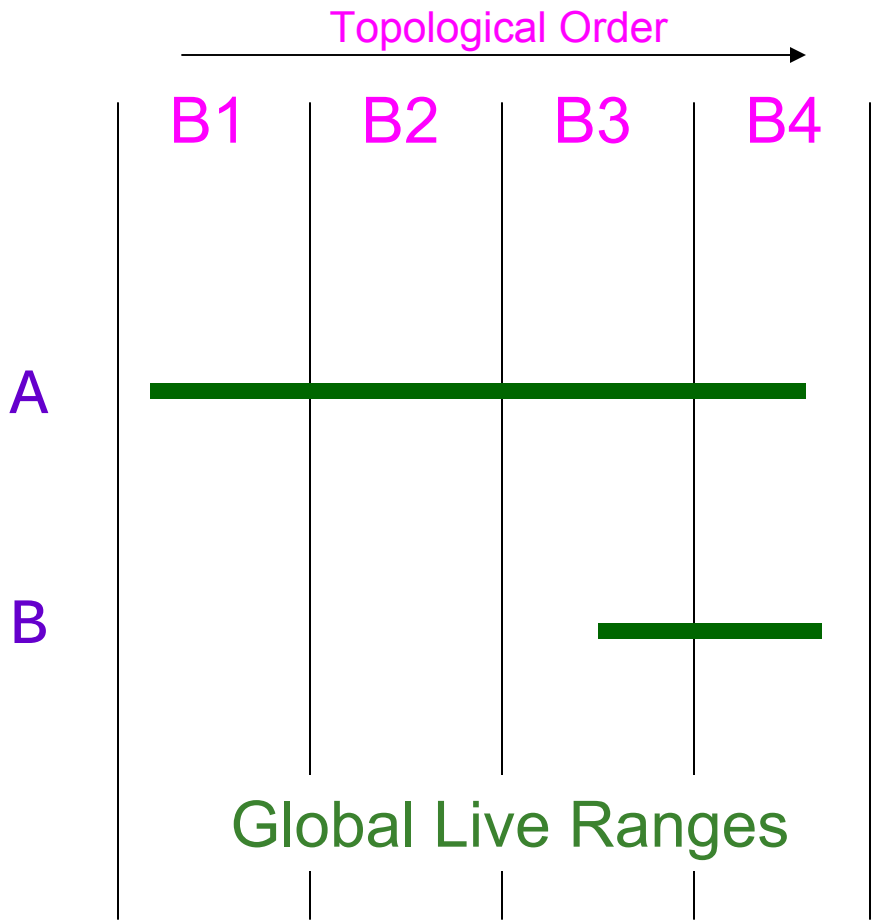
- A topological order linearizes a partial order

Global Linear Scan RA

- Ignoring back-edges, perform a topological sort of the basic blocks using the CFG
- Compute the live range over the entire topological order
- Treat the blocks in topological order as a single large block and perform linear scan

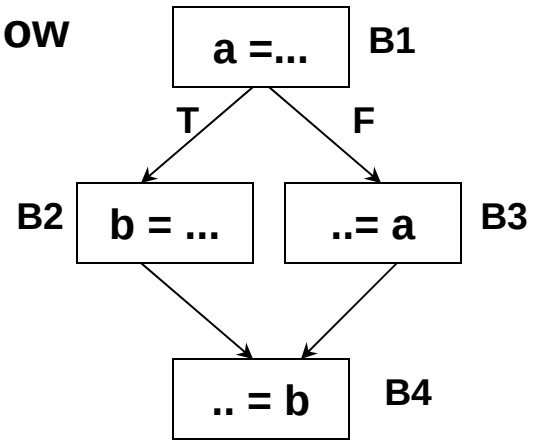
Global Live Ranges





Simple Example of Global Register Allocation

Control Flow Graph



Live range of a = {B_1, B_3}

Live range of b = {B_2, B_4}

No interference! a and b can be assigned to the same register