

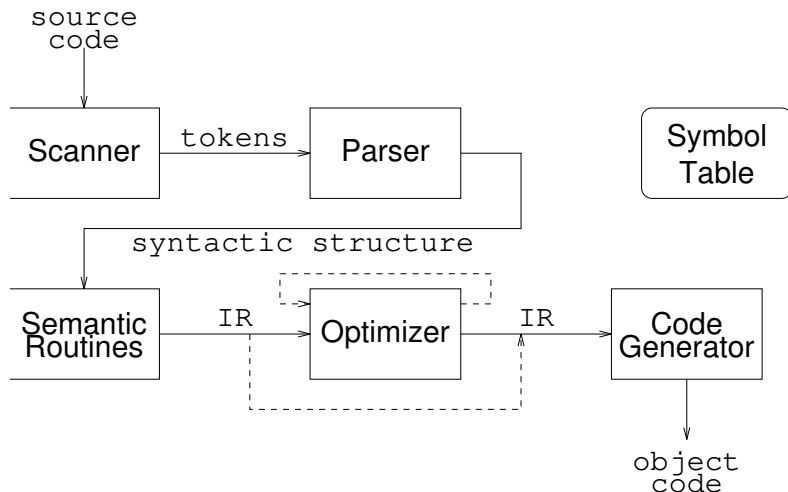
Liveness Analysis and Register Allocation

KC Sivaramakrishnan

IIT Madras

Monsoon 2026

The Compiler



Challenges in the back end

- **Input:** intermediate representation (IR).
- **Target:** a machine with a specific instruction set, constraints, and a finite number of registers.
- **Instruction selection:** choose machine instructions that implement the IR.
- **Register allocation:** decide which temporaries to keep in registers and which to store in memory.

Maps groups of IR instructions to one or more machine instructions.

Why not translate each IR instruction independently?

- This is easy if we do not care about efficiency.
- There may be several choices, such as `add` or `inc`.
- The best choice may depend on the surrounding instructions. A machine instruction may implement several IR operations together.

Instruction selection: an array load

Load $A[i]$, where each element occupies 4 bytes. Assume the array base is in `rdi` and the index `i` is in `rsi`.

```
t1 = i * 4
t2 = base + t1
x   = load32(t2)
```

Translate separately

```
mov rax, rsi
shl rax, 2
add rax, rdi
mov eax, [rax]
```

Select the whole pattern

```
mov eax, [rdi + rsi*4]
```

x86-64 scaled-index addressing folds the multiply and add into the load.

Intel syntax; result in `eax`. Assume `t1` and `t2` are not used elsewhere.

Register allocation

IR can contain arbitrarily many temporaries; the machine has a finite number of registers.

- Keep frequently used values in registers.
- When registers are insufficient, store values in memory (spilling). This adds loads and stores.
- Related decisions: register allocation, register assignment, spill location, and coalescing.

The Memory Hierarchy

Level	Approximate latency
Registers	~ 1 cycle
L1 cache hit	A few cycles
L2 cache hit	About a dozen cycles
L3 cache hit	Tens of cycles
Main memory (DRAM)	Hundreds of cycles
SSD	Tens to hundreds of μs
HDD	Several ms

Illustrative values; actual latency depends on hardware and workload. The register row refers to a simple integer operation on register operands.

Managing the Memory Hierarchy

- Most programs are written as if there are only two kinds of memory: main memory and disk.
- Programmer is responsible for moving data from disk to memory (i.e. file I/O).
- Hardware is responsible for moving data between memory and caches.
- Compiler is responsible for moving data between memory and registers.

Managing the Memory Hierarchy

- Note that there is an order of magnitude difference between register/cache access and main memory access.
- Hence, it is very important to
 - Manage caches properly.
 - Manage registers properly.
- Cache behaviour is in general unpredictable (actually undecidable).
 - Hence, as a compiler designer, managing registers properly becomes even more important!

Register allocation:

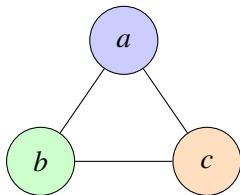
- have value in a register when used
- limited resources
- can affect the instruction choices
- spilling inserts stores and reloads; their placement affects performance
- optimal allocation is difficult: the general spill-free assignment problem is NP-complete

Register allocation as graph colouring

Build an interference graph:

- A vertex represents a temporary.
- An edge joins temporaries that must hold different values at the same time.
- A colour represents a physical register.

Adjacent vertices must receive different colours.



Here all three pairs interfere.

Three colours $\Leftrightarrow$ three registers.

Two registers cannot hold all three without spilling.

Why is register allocation NP-complete?

Decision problem: can all temporaries be assigned to k registers without spilling?

In the general interference-graph model:

- **In NP:** given an assignment, check that every temporary has a register and every interference edge has different registers at its ends. This takes polynomial time.
- **NP-hard:** graph k -colouring reduces to register assignment. General programs can realise arbitrary interference graphs; a valid assignment gives a valid colouring.

Graph k -colouring $\leq_p$ Spill-free register assignment

Assume each temporary is assigned one register throughout its lifetime.

Basic blocks

A graph representation of intermediate code.

Each basic block is a maximal sequence of 3-address code instructions with following properties:

- The flow of control can only enter the basic block through the first instruction in the block.
- No jumps into the middle of the block.
- Control leaves the block without halting / branching (except may be the last instruction of the block).

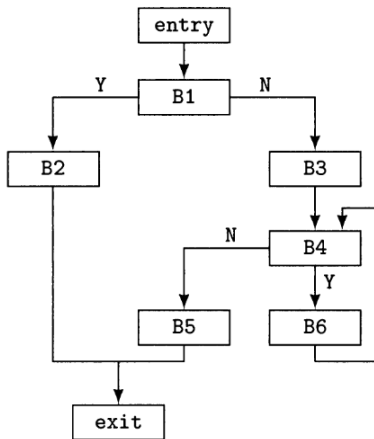
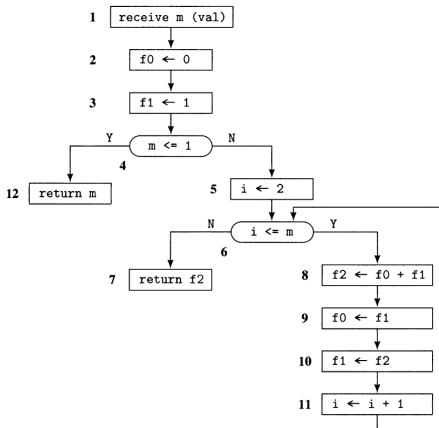
The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

Example

```
unsigned int fib(m)
{
    unsigned int m;
    { unsigned int f0 = 0, f1 = 1, f2, i;
      if (m <= 1) {
          return m;
      }
      else {
          for (i = 2; i <= m; i++) {
              f2 = f0 + f1;
              f0 = f1;
              f1 = f2;
          }
          return f2;
      }
    }
}
```

```
1      receive m (val)
2      f0 ← 0
3      f1 ← 1
4      if m <= 1 goto L3
5      i ← 2
6  L1:  if i <= m goto L2
7      return f2
8  L2:  f2 ← f0 + f1
9      f0 ← f1
10     f1 ← f2
11     i ← i + 1
12     goto L1
13  L3:  return m
```

Example - flow chart and control-flow



- The high-level abstractions might be lost in the IR.
- Control-flow analysis can expose control structures not obvious in the high level code.

CFG - Control flow graph

Definition:

- A rooted directed graph $G = (N, E)$ for a single-entry function. N contains its basic blocks and two special nodes: `entry` and `exit`.
- An edge connects basic blocks b_1 and b_2 if control can pass directly from b_1 to b_2 .
- An edge connects the `entry` node to the first basic block.
- An edge connects each final basic block (for example, one ending in a return) to the `exit` node.

Liveness analysis

Problem:

- IR contains an unbounded number of temporaries
- machine has bounded number of registers

Approach:

- temporaries with disjoint live ranges can map to same register
- if not enough registers then spill some temporaries
(i.e., keep them in memory)

The compiler must perform liveness analysis for each temporary:

It is live if it holds a value that may be needed in future

Example

```
 $L_1 :$    $a \leftarrow 0$   
        $b \leftarrow a + 1$   
        $c \leftarrow c + b$   
       if  $c < N$  goto  $L_1$   
       return  $c$ 
```

a and b can be allocated to the same register

Example

$L_1 :$ $a \leftarrow 0$
 $b \leftarrow a + 1$
 $c \leftarrow c + b$
if $c < N$ goto L_1
return c

$L_1 :$ $r_1 \leftarrow 0$
 $r_1 \leftarrow r_1 + 1$
 $r_2 \leftarrow r_2 + r_1$
if $r_2 < N$ goto L_1
return r_2

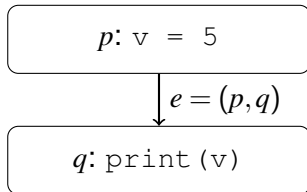
Liveness analysis

Gathering liveness information is a form of data flow analysis operating over the CFG:

- We will treat each statement as a different basic block.
- liveness of variables “flows” through the edges of the graph
- assignments define a variable, v :
 - $def(v)$ = set of graph nodes that define v
 - $def[n]$ = set of variables defined by n , where n is a BB.
- occurrences of v in expressions use it:
 - $use(v)$ = set of nodes that use v
 - $use[n]$ = set of variables used in n , where n is a BB.

Liveness on an edge

- v is live on edge $e = (p, q)$ if, starting at q , some directed path reaches a use of v before v is redefined.



- On edge e , p has already executed; q has not.
- v is live on e : the value produced by p is used at q .
- Start at q and look for a use before another definition of v . The definition at p produces the value we are tracking.

Definitions: live-in and live-out

- v is live-in at node n if live on any of n 's in-edges
- v is live-out at n if live on any of n 's out-edges
- $v \in use[n] \Rightarrow v$ live-in at n (recall: each statement is its own basic block).
- v live-out at $n \wedge v \notin def[n] \Rightarrow v$ live-in at n

Liveness analysis

Liveness is a backward dataflow analysis over the control flow graph: information flows from successors to predecessors.

Define:

$$\begin{aligned} in[n] &= \text{variables live-in at } n \\ out[n] &= \text{variables live-out at } n \end{aligned}$$

Then:

$$\begin{aligned} out[n] &= \bigcup_{s \in succ(n)} in[s] \\ succ[n] = \phi &\Rightarrow out[n] = \phi \end{aligned}$$

Note:

$$\begin{aligned} in[n] &\supseteq use[n] \\ in[n] &\supseteq out[n] - def[n] \end{aligned}$$

Thus:

$$in[n] = use[n] \cup (out[n] - def[n])$$

Iterative solution for liveness

N : Set of nodes of CFG;

foreach $n \in N$ **do**

$in[n] \leftarrow \phi$;

$out[n] \leftarrow \phi$;

end

repeat

foreach $n \in \text{Nodes}$ **do**

$in'[n] \leftarrow in[n]$;

$out'[n] \leftarrow out[n]$;

$in[n] \leftarrow use[n] \cup (out[n] - def[n])$;

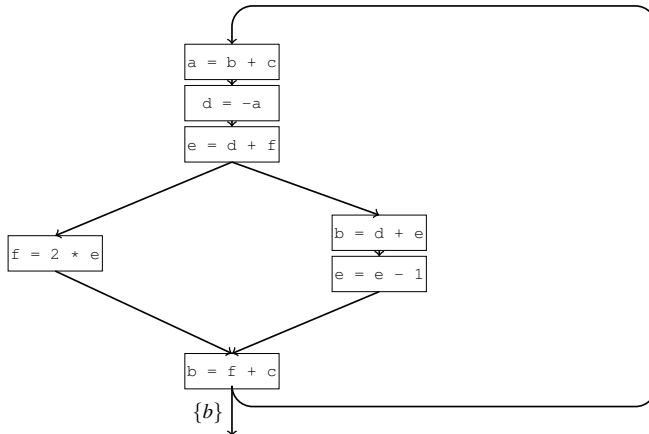
$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$;

end

until $\forall n, in'[n] = in[n] \wedge out'[n] = out[n]$;

- At a fixed point, applying the equations changes no set.
- Starting with empty sets gives the least fixed point: the smallest sets (by inclusion) satisfying all the equations together.

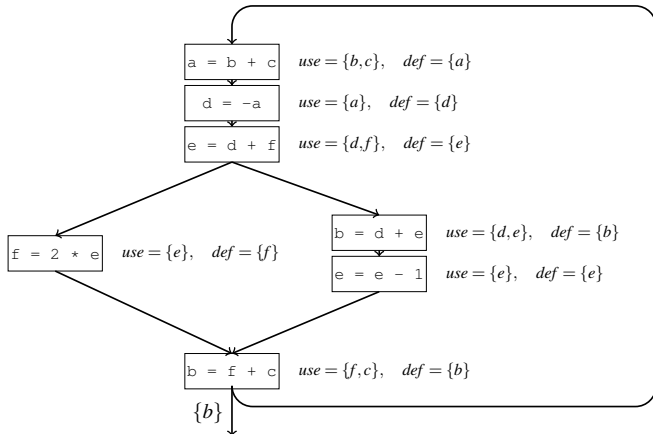
Example: one block per statement



$$out[n] = \bigcup_{s \in succ(n)} in[s]$$

$$in[n] = use[n] \cup (out[n] - def[n])$$

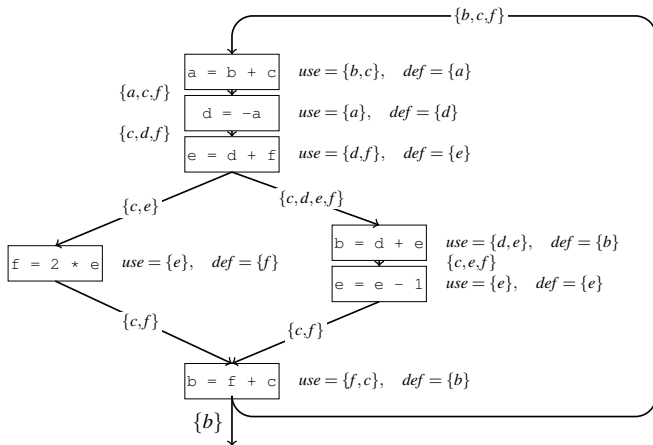
Example: use and def sets



$$out[n] = \bigcup_{s \in succ(n)} in[s]$$

$$in[n] = use[n] \cup (out[n] - def[n])$$

Example: live-in and live-out sets



$$out[n] = \bigcup_{s \in succ(n)} in[s]$$

$$in[n] = use[n] \cup (out[n] - def[n])$$

- should order computation of inner loop of the data-flow analysis algorithm to follow the “flow”
- liveness flows backward along control-flow arcs, from out to in
- nodes can just as easily be basic blocks to reduce CFG size
- could do one variable at a time, from uses back to defs, noting liveness along the way

Iterative solution for liveness

Complexity: for an input program of size N

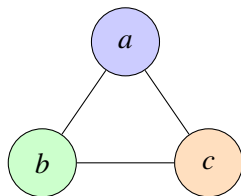
- At most N CFG nodes and at most N variables.
Each *in/out* set contains at most N variables,
so each set operation takes $O(N)$ time.
 - One pass (the **for** loop) visits every node and performs a constant number of set operations per node.
 $\Rightarrow O(N^2)$ time per pass.
 - Sets only grow: variables are added, never removed.
Across all *in* and *out* sets, at most $2N^2$ entries can be added.
Every pass before convergence adds at least one.
 $\Rightarrow O(N^2)$ passes of the **repeat** loop.
- $\Rightarrow$ Worst-case time: $O(N^2) \times O(N^2) = O(N^4)$.
- A good node order can reduce this to 2–3 passes.
In practice: $O(N)$ with constant-time set operations, or $O(N^2)$ with $O(N)$ set operations.

Least fixed points

- There is often more than one solution for a given dataflow problem.
 - For example, if a variable x is never used or defined, then adding x to the *in* and *out* sets of every basic block is also a valid solution.
- We want the “smallest” solution: the least fixed point.
 - The iterative algorithm computes this least fixed point.
- Any solution to dataflow equations is a conservative approximation:
 - v has some later use downstream from n
 $\Rightarrow v \in out(n)$
 - but not the converse
- Conservatively assuming a variable is live does not break the program; just means more registers may be needed.
- Assuming a variable is dead when really live will break things.

Register Interference Graph

- An undirected graph
 - A node for each temporary/variable.
 - An edge between t_1 and t_2 if they are live simultaneously at some point in the program.



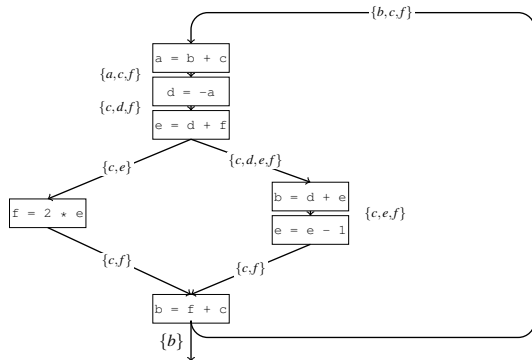
All three pairs interfere.

- Two temporaries can be allocated to the same register if there is no edge connecting them.
- The register interference graph extracts exactly the information needed to perform legal register assignments.
- Also gives a global picture (i.e. over the entire flow graph) of the register requirements.

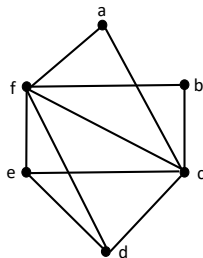
Example: Register Interference Graph

Connect every pair of variables that appear together in a live set.

Program and live sets



Interference graph

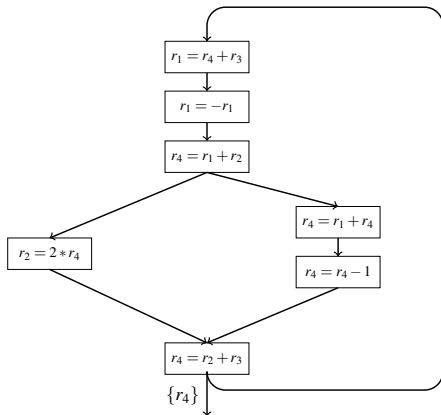
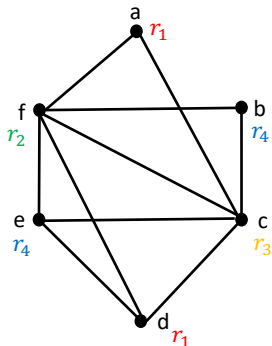


- Live set $\{b, c, f\}$ gives edges $b-c$, $b-f$, and $c-f$.
- b and d can be assigned the same register.
- How many registers? Same as the number of colors required for coloring the graph.

Register allocation - by Graph coloring

- Step 1:
 - Select target machine instructions assuming infinite registers (temps).
 - If a instruction requires a special register – replace that temp with that register.
- Step 2:
 - Construct the interference graph.
 - Solve the register allocation problem by coloring the graph.
 - A graph is said to be colored if each pair of neighboring nodes have different colors.

Example: Colored RIG



Computing Graph Colorings

- How do we compute graph colorings?
- The problem is NP-Hard. No efficient algorithms are known.
 - Solution: We will use heuristics.
- A coloring may not exist for a given number of registers/colors.
 - Solution: We will use systematic spilling.

Graph Coloring Heuristic

- Observation:
 - Pick a node t with fewer than k neighbours in RIG.
 - Eliminate t and its edges from RIG.
 - If the resulting graph is k -colorable, then so is the original graph.
- Why?
 - Let $c_1, c_2, \dots, c_n$ be the colors assigned to neighbours of t in the reduced graph.
 - Since $n < k$, we can pick a color for t among k colors that is different from those of its neighbours.

Graph coloring - a simplistic approach

Input: G - the interference graph, k - number of colors

repeat

repeat

 Remove a node n and all its edges from G , such that degree of n is less than K ;

 Push n onto a stack;

until G has no node with degree less than k ;

 // G is either empty or all of its nodes have degree $\geq k$

if G is not empty **then**

 Take one node m out of G , and mark it for spilling;

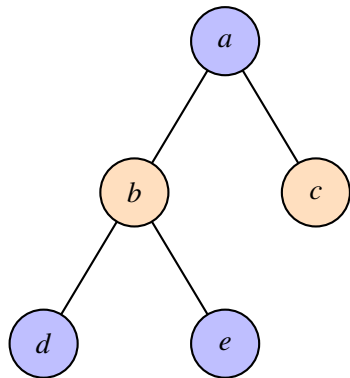
 Remove all the edges of m from G ;

end

until G is empty;

Take one node at a time from the stack and assign a non conflicting color.

Example 1, available colors = 2



Select: restore d

Neighbour b is orange. Assign blue (r_1). No spill is needed.

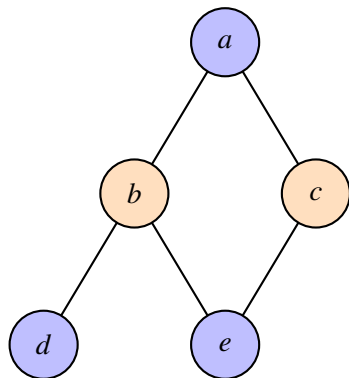
Stack (top at right):

$\langle \rangle$

Removal order: d, e, b, c, a . **Colouring order:** a, c, b, e, d .

Faded nodes and edges have been removed from the current graph.

Example 2, available colors = 2



Two colours suffice

Give d blue: its neighbour b is orange. No spill is needed.

After removing d , every remaining node has degree 2, so the simplistic algorithm gets stuck and chooses an unnecessary spill.

Degree $< k$ guarantees safe removal; it is not necessary for k -colourability.

Graph coloring - Kempe's heuristic

- Algorithm dating back to 1879.
- Also called 'optimistic coloring'.

Input: G - the interference graph, K - number of colors

repeat

repeat

 Remove a node n and all its edges from G , such that degree of n is less than K ;

 Push n onto a stack;

until G has no node with degree less than K ;

 // G is either empty or all of its nodes have degree $\geq K$

if G is not empty **then**

 Take one node m out of G .;

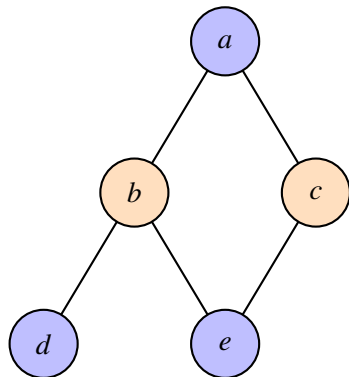
 push m onto the stack;

end

until G is empty;

Take one node at a time from the stack and assign a non conflicting color (if possible, else spill).

Example 2 (revisited): optimistic colouring



Select: restore d

Neighbour b is orange. Assign blue.

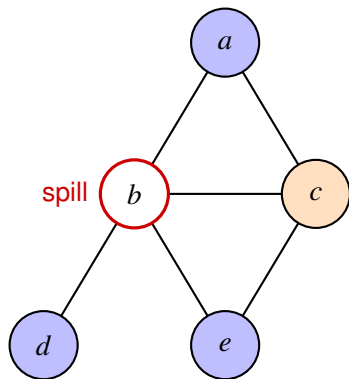
The optimistic choice succeeded: b had two neighbours, but they used only one colour.

Stack (top at right):

$\langle \rangle$

b^* : removed optimistically; spill only if no colour is available when restored.

Example 3: two colours are insufficient



A spill is unavoidable

Assign d blue; b has no register.

The triangle a, b, c needs three colours. With only two registers, a spill-free colouring is impossible.

Stack (top at right):

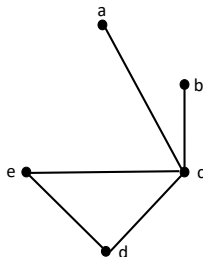
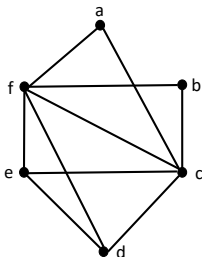
$\langle \rangle$

Red outline: selected for spilling, not a third register colour.
Next: rewrite the code for the spill and recompute liveness.

- We need to generate extra instructions to load variables from the stack and store them back.
- The load and store may require registers again:
 - Naive approach: Keep a separate register (wasteful).
 - Rewrite the code - by introducing a temporary; rerun the liveness + register-allocation
(Note: the new temp has much smaller live range).

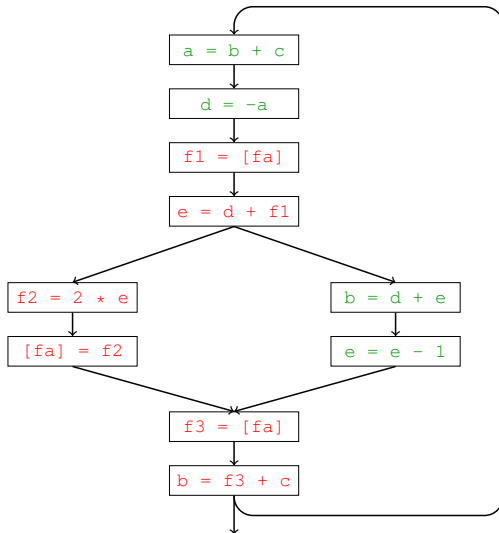
Example: Spilling

Suppose we have 3 registers and decide to spill f .



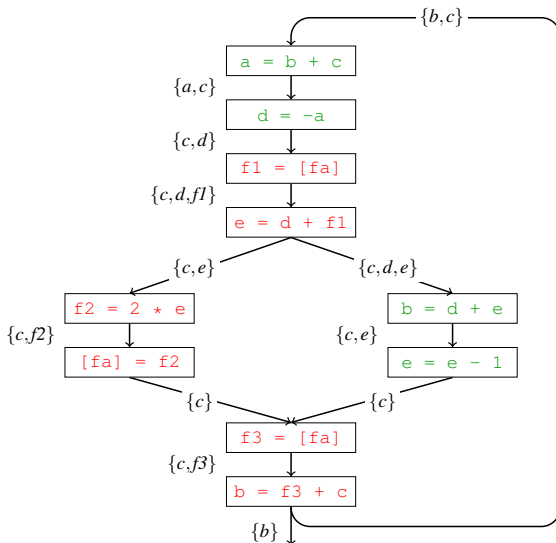
- Reserving one scratch register for spill loads/stores leaves only **2 registers** for the remaining variables.
- The RHS graph is **not 2-colourable**: c, d, e form a triangle.
- Instead, introduce short-lived temporaries for spill accesses and rerun allocation using all 3 registers.

Example: Rewrite for spilled variable



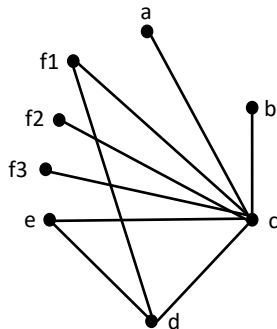
`[fa]` is the spill slot for `f`. **Green: unchanged. Red: inserted or rewritten.**

Example: Recomputing liveness after rewrite



f is now in memory; $f1, f2, f3$ have short live ranges.

Example: RIG after rewrite



Homework: Find a register allocation using 3 registers, and rewrite the program using the allocated registers.

Criteria for spilling

Choose a live range: balance pressure relief against runtime cost.

- **Random**
 - Cheap baseline; ignores interference and runtime cost.
- **Maximum degree**
 - Removes the most edges, helping neighbours get registers; ignores how often the chosen variable is used.
- **Longest live range**
 - Frees a register over a large region; a long range need not have many uses.
- **Shortest live range**
 - May benefit from nearby cached accesses, but frees a register only briefly. Short lifetime does not guarantee cache hits or low cost.
- **Lowest estimated spill cost**
 - Weight extra load/store costs by execution frequency (from profiles or loop/branch estimates).
 - A reload inside a loop may execute 1,000 times; one on a rare branch may not execute at all.

More details: textbook, Chapter 8, Section 8.8.2.

Register allocation - Linear scan

Register allocation is **expensive**.

- Many algorithms use heuristics for graph coloring.
- Allocation may take time quadratic in the number of live intervals.

Not suitable

- Online compilers – need to generate code quickly. e.g. JIT compilers.
- Sacrifice efficient register allocation for compilation speed.

Massimiliano Poletto and Vivek Sarkar. 1999.

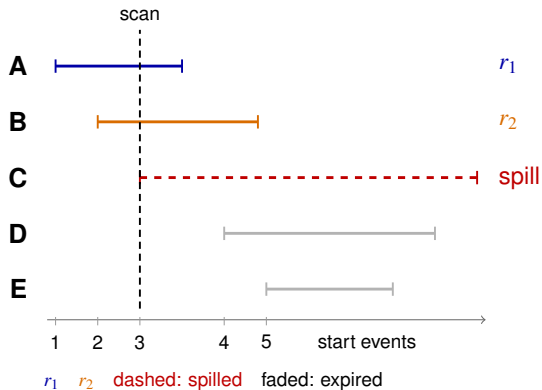
“Linear Scan Register Allocation.”

ACM Transactions on Programming Languages and Systems **21**(5), 895–913 (September 1999).

DOI: 10.1145/330249.330250.

Self-study: Section 4 of the above paper.

Live intervals and the active set



Live interval

A span from a value's first live point to its last, in the chosen instruction order.

Active set

Intervals covering the current scan position that are assigned registers.

After processing C :

$active = \langle A : r_1, B : r_2 \rangle$.

C overlaps this position but is spilled, so it is **not active**.

- A live interval may include gaps where the value is not live.
- Labels 1–5 mark interval starts in scan order, not instruction numbers.

Linear scan: the idea

Process **live intervals** in increasing order of their start points.

① **Expire finished intervals.**

- Remove them from the **active set** and release their registers.

② **Use a free register, if one is available.**

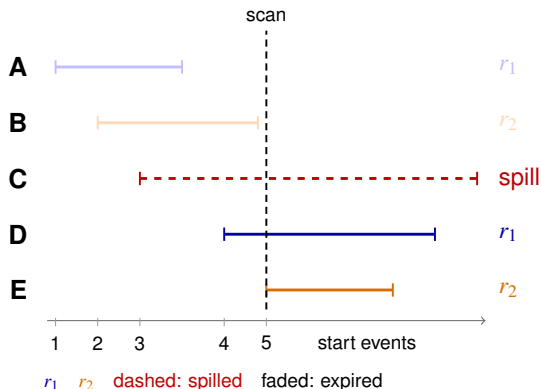
- Assign it to the new interval and add that interval to the **active set**.
- The **active set** contains only intervals currently assigned registers.

③ **Otherwise, spill the interval that ends furthest in the future.**

- Keep the **active set** sorted by end point.
- Compare the new interval with the active interval that ends last.
- Spill the new interval, or give it the evicted interval's register.

Poletto and Sarkar (1999), Section 4 and Fig. 1. doi:10.1145/330249.330250

Linear scan example: 2 registers



5. Start E

B has ended: release r_2 and assign it to E .

Active (end order):

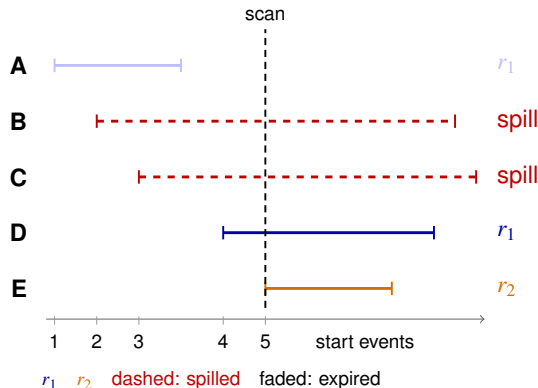
$\langle E : r_2, D : r_1 \rangle$

Spilled: C

B has expired: reuse its register r_2 for E . Only C is spilled.

Linear scan: evicting an active interval

A different program: the live range of B is longer.



Give B 's register to E

Mark B for spilling and assign r_2 to E .

Active (end order):

$\langle E : r_2, D : r_1 \rangle$

Spilled: B, C

After choosing the spills, rewrite the code with loads and stores for the spilled variables.

Linear Scan algorithm - analysis

- Each live range gets either a register or a spill location.
- Note: The number of overlapping intervals changes only at the start and end points of an interval.
- Live intervals are stored in a list that is sorted in order of increasing start point.
- The active list is kept sorted in order of increasing end point. Adv: need to scan only those intervals (+1 at most) that have to be removed.
- Complexity: $O(V)$ – if number of registers is assumed to be a constant. Else? $O(V \log R)$
 - The $\log R$ factor comes from maintaining the active set (at most R intervals) in end-point order using a balanced search tree.

Summary

- **Instruction selection** maps IR operations to machine instructions; combining operations can avoid unnecessary work.
- **Liveness analysis** propagates information backwards through the CFG to a fixed point: which values may still be needed?
- **Graph colouring** assigns different registers to interfering values. Simplification and optimistic colouring guide the allocation.
- **Spilling** trades registers for memory accesses. Choose low-cost candidates, rewrite the code, and recompute liveness.
- **Linear scan** processes live intervals in start order, reuses registers from expired intervals, and spills the interval that ends last when needed.

The trade-off: allocation time versus the runtime cost of the generated code.