CS 335: Register Allocation

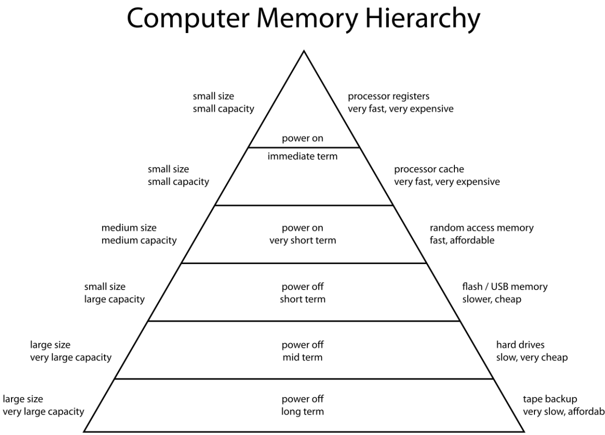
Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

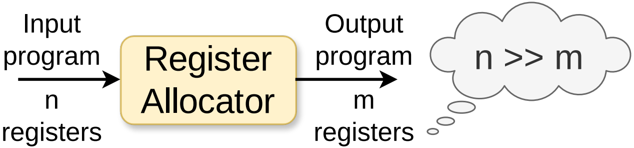
Impact of Register Operands

- Instructions involving register operands are faster than those involving memory operands
  - Code is often smaller and hence is faster to fetch
- Efficient utilization of registers is important
  - Number of general-purpose registers is limited (e.g., 16–32 64-bit GPRs)



Goals in Register Allocation

- All variables are not used (or live) at the same time
- Register allocator in a compiler helps answer the following questions
  (i) Which values should reside in registers?
  (ii) Which register should hold each of those values?
- At each program location, values stored in virtual registers in the IR are mapped to physical registers



Goals in Register Allocation

- Given that programs spend most of their time in loops, it is natural to store values in innermost loops in registers
- **Register pressure** measures the availability of free registers
  - High register pressure implies that a large fraction of registers are in use
- When no registers are available to store a computation, the contents of one of the in-use registers must be stored in memory
  - This is called **register spilling**, and requires generating load and store instructions
  - Spilling increases execution time and code size
- The goal in register allocation is to **minimize the impact of spills**, especially for performance-critical code

Register Allocation and Assignment

Register Allocation

- Maps an unlimited namespace onto the register set of the target machine
- Register-to-register model: map virtual registers to the physical register set and spill values that do not fit in the physical register set
- Memory-to-memory model: map a subset of memory locations to a set of physical registers

Register Assignment

- Assumes that allocation has already been performed
- Maps an allocated name set to the physical registers of the target machine

Challenges in Register Allocation

(i) Architectures provide **different register classes**

- General purpose registers, floating-point registers, predicate and branch target registers
- General-purpose registers may be used for floating-point register spills, which implies an order for allocation
- Registers may be aliased, cannot use a register and its constituent registers at the same time
    - x86 has 32-bit registers whose lower halves are used as 16-bit or 8-bit registers
    - Similar for vector registers like zmm, ymm, and xmm
- If different register classes overlap, the compiler must allocate them together

(ii) Architecture **calling conventions place** more **constraints** on register usage

- For example, PowerPC requires parameters to be passed in R3-R10 and the return is in R3

Register Allocation Problem

- General formulation of the problem is NP-Complete
  - For example, register allocation for a set of BBs, multiple control flow paths, multiple data types, and non-uniform cost of memory access complicate the analysis
  - Optimal allocation can be done in polynomial time for very restricted versions with a single BB and with one data type

More about Register Allocation

- Types of allocation

  local over a BB

  global over a whole function

  inter-procedural across function boundaries traversed via call graph

- Backend includes instruction selection, register allocation, and instruction scheduling
  - Performing allocation first restricts the movement of code during scheduling, not a good idea
  - Scheduling instructions first cannot handle spill code introduced during allocation
  - Possible order: selection → scheduling → allocation → scheduling

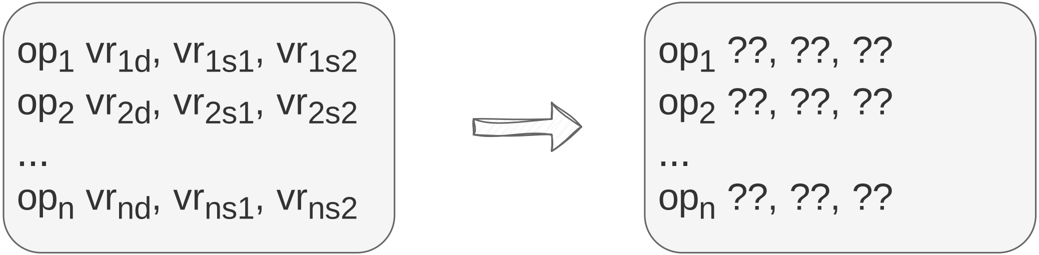
Local Register Allocation

Frequency count and variable usage

Local Register Allocation

Assumptions

- Considers only a single basic block (BB)
- Loads values from memory at the start of a BB and stores to memory at the end of the BB
- IR makes use of many virtual registers
- Target machine has a **uniform** class of *k* general-purpose registers



$op_1\ vr_{1d},\ vr_{1s1},\ vr_{1s2}$
$op_2\ vr_{2d},\ vr_{2s1},\ vr_{2s2}$
...
$op_n\ vr_{nd},\ vr_{ns1},\ vr_{ns2}$

op_1 ??, ??, ??
op_2 ??, ??, ??
...
op_n ??, ??, ??

Top-Down Allocation with Frequency Counts

- Count the **frequency of occurrence** of virtual registers
- Map virtual registers to physical registers in **descending** order of frequency
- If the BB uses fewer than k virtual registers, then mapping is trivial
- A few registers ($F \approx 2 - -4$) registers are reserved to execute spill code
- Assign the top $k - F$ virtual registers to physical registers
- Rewrite the code and replace virtual registers with physical registers
- For unassigned virtual registers, generate code sequence to spill code using the F reserved registers

Example of Top-Down Allocation

Assume there are three physical registers

Usage Count
$R_a = 4$
$R_b = 3$
$R_c = 2$
$R_d = 2$
$R_e = 2$
$R_f = 2$
$R_g = 2$
$R_h = 2$

	Liveness Information
LD R_a, 1028	
	R_a
MOV R_b, R_a	
	R_a R_b
MUL R_c, R_a, R_b	
	R_a R_b R_c
LD R_d, x	// Spill R_c based on usage count
	R_a R_b R_c R_d
SUB R_e, R_d, R_b	// Reuse R_b
	R_a R_c R_e
LD R_f, z	// Reuse R_d
	R_a R_c R_e R_f
MUL R_g, R_e, R_f	// Reuse R_e
	R_a R_c R_g
SUB R_h, R_g, R_c	// Restore R_c, reuse R_g
	R_a
MOV R_a, R_h	

Example of Top-Down Allocation

```
LD R_a, 1028
MOV R_b, R_a
MUL R_c, R_a, R_b
LD R_d, x
SUB R_e, R_d, R_b
LD R_f, z
MUL R_g, R_c, R_f
SUB R_h, R_g, R_c
MOV R_a, R_h
```

$\xrightarrow[\text{spill}]{\text{Generate}}$

```
LD R_a, 1028
MOV R_b, R_a
MUL R_c, R_a, R_b
ST off_c(R_SP), R_c
LD R_d, x
SUB R_e, R_d, R_b
LD R_f, z
MUL R_g, R_e, R_f
LD R_c, off_c(R_SP)
SUB R_h, R_g, R_c
MOV R_a, R
```

$\xrightarrow[\text{assignment}]{\text{Register}}$

```
LD R_1, 1028
MOV R_2, R_1
MUL R_3, R_1, R_2
ST off_c(R_SP), R_3
LD R_3, x
SUB R_2, R_3, R_2
LD R_3, z
MUL R_2, R_2, R_3
LD R_3, off_c(R_SP)
SUB R_2, R_2, R_3
MOV R_1, R_2
```

- Top-down local allocation allocates a physical register to one virtual register for the **entire BB**
- Allocation can be suboptimal if variables show **phased** behavior (e.g., R_a)
  - A variable that is heavily used in the first half of the BB and not used in the second half still stays in the physical register

Bottom-Up Allocation Based on Variable Usage

- Iterate over the instructions in the BB and make decisions based on variable **usage** (not count)
- Assumes that the physical registers are initially empty and places them on a free list
- Satisfies demand for registers from the free list until that list is exhausted
- If the free list is empty, spill a variable to memory and reuse the register
    - Spill the variable whose **next use is farthest** in the future

Bottom-Up Allocation

- Assume that registers are grouped in classes (e.g., general-purpose, floating-point)
  - Size: # of physical registers,
  - Name: virtual register name,
  - Next: distance to next reuse,
  - Free: flag to indicate whether currently in use,
  - Stack: free physical registers

```
struct RegClass {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

Helper Functions in the Bottom-Up Algorithm

```
Ensure(vr, regClass)
  if vr is already in regClass
    pr = physical register for vr
  else
    pr = Allocate(vr, regClass)
    emit code to move vr into pr
  return pr

Free(pr, regClass)
  // 0 ≤ i < Size
  regClass.Name[i] = null
  regClass.Next[i] = -1
  regClass.Free[i] = true
  push(regClass, pr)
```

```
Allocate(vr, regClass)
  if regClass.StackTop >= 0
    // free register available
    i = pop(regClass)
  else
    // Check for farthest use
    i = j s.t. $\forall_{j \in Size}$ max(regClass.Next[j])
    // Emit spill code
    store contents of j
  regClass.Name[i] = vr
  regClass.Next[i] = -1
  regClass.Free[i] = false
  return i
```

Bottom-Up Algorithm

```
for each instruction i = {1 ... n} in BB
  // Instruction i: $op_i$ $vr_{i_d}$ $vr_{i_{s1}}$, $vr_{i_{s2}}$
  rx = Ensure($vr_{i_{s1}}$, regClass($vr_{i_{s1}}$))
  ry = Ensure($vr_{i_{s2}}$, regClass($vr_{i_{s2}}$))
  if $vr_{i_{s1}}$ is not needed after i
    Free(rx, regClass(rx))
  if $vr_{i_{s2}}$ is not needed after i
    Free(ry, regClass(ry))
  rz = Allocate($vr_{i_d}$, regClass($vr_{i_d}$))
  rewrite i as $op_i$ rz, rx, ry
  if $vr_{i_{s1}}$ is needed after i
    regClass.next[rx]= Dist($vr_{i_{s1}}$)
  if $vr_{i_{s2}}$ is needed after i
    regClass.next[ry]= Dist($vr_{i_{s2}}$)
  regClass.next[rz]= Dist($vr_{i_d}$)
```

Behaviour with Bottom-Up Allocation

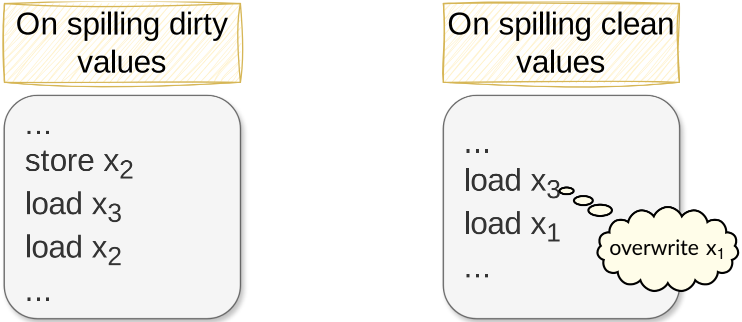
Assume there are three physical registers

	Liveness Information						
LD R_a, 1028							
	R_a						
MOV R_b, R_a							
	R_a	R_b					
MUL R_c, R_a, R_b							
	R_a	R_b	R_c				
LD R_d, x	// Spill R_a based on farthest use						
	R_a	R_b	R_c	R_d			
SUB R_e, R_d, R_b	// Reuse R_b						
	R_a		R_c		R_e		
LD R_f, z	// Reuse R_d						
	R_a		R_c		R_e	R_f	
MUL R_g, R_e, R_f	// Reuse R_e						
	R_a		R_c				R_g
SUB R_h, R_g, R_c	// Restore R_a, reuse R_g						
	R_a						
MOV R_h, R_a							

Challenges in Bottom-Up Allocation

- A store on a spill is unnecessary if the data is clean
  - That is, the register contains a constant value or a return from a load
- A spill should be stored only if the data is dirty

- Assume a two-register machine: values x_1 is clean and x_2 is dirty
- Assume the reference stream for the rest of the BB is $x_3x_1x_2$



On spilling dirty values

```
...
store x2
load x3
load x2
...
```

On spilling clean values

```
...
load x3
load x1
...
```

overwrite x_1

Challenges in Bottom-Up Allocation

- A store on a spill is unnecessary if the data is clean
  - That is, the register contains a constant value or a return from a load
- A spill should be stored only if the data is dirty

- Assume a two-register machine: values x_1 is clean and x_2 is dirty
- Assume the reference stream for the rest of the BB is $x_3 x_1 x_3 x_1 x_2$

On spilling clean values

...
load x_3
load x_1
load x_3
load x_1
...

On spilling dirty values

...
store x_2
load x_3
load x_2
...

Global Register Allocation

Global Register Allocation

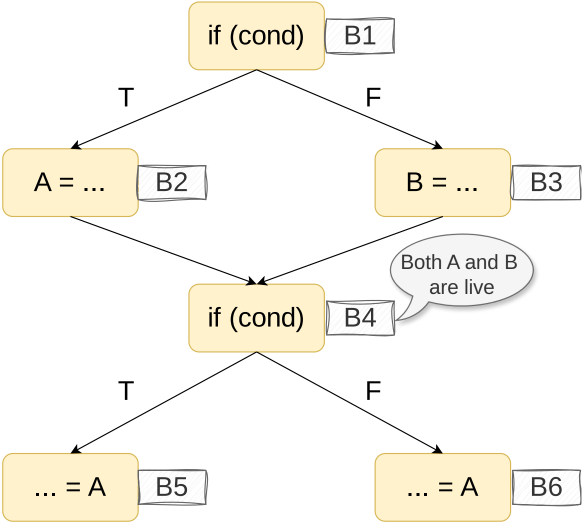
- Scope is either multiple BBs or a whole procedure
- **Decision problem**: Given an input program in IR form (e.g., CFG) and a number *k*, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads and stores are introduced, and at most *k* registers are used?
- Fundamentally a more complex problem than local register allocation
  - Need to consider def-use across multiple blocks
  - Cost of spilling may not be uniform because it depends on the execution frequency of the block where a spill happens

Live Ranges

- Rather than simply allocating a variable to a register throughout an entire sub-program (i.e., the BBs under analysis), it is more profitable to split variables into live ranges
- A variable is **live** at a point *p* in the CFG if there is a use of the variable in the path from *p* to the end of the CFG
- A **live range** for a variable is the **smallest set of program points** at which it is live
  - A variable's live range starts at the point in the code where the variable receives a value and ends where that value is used for the last time
  - Live ranges can be tracked in terms of individual instructions or BBs
  - A single variable may be represented by many live ranges
  - Two overlapping live ranges for the same variable must be merged
    - Merged live ranges are also called webs
- Two variables interfere or **conflict** if their live ranges intersect (i.e., both are live)

Example of Live Ranges

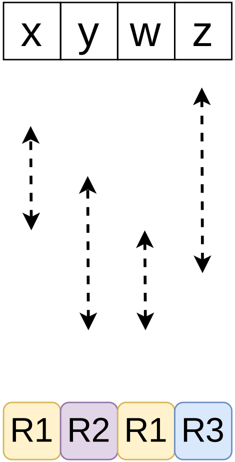
```
if (cond) {
  A = …
} else {
  B = …
}
if (cond) {
  … = A
} else {
  … = B
}
```



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

Utility of Live Ranges

```
z = 1
x = 2 * x
y = 3 * z
w = x + y
print y+z
x = y * w
```



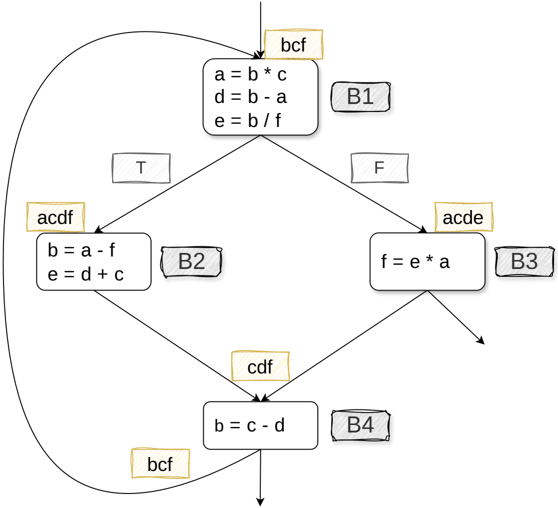
x	y	w	z
R1	R2	R1	R3

x's and w's live ranges do not overlap! They can therefore be assigned to the same register.

Global Register Allocation Based on Usage Counts

- Heuristic method to allocate registers for most **frequently-used** variables
- Requires information about liveness of variables at the entry and exit of each BB in the loop body L
- After v is computed into a register (i.e., defined), it stays in that register until the end of the BB and all further references are to that register
- For every usage of a variable v in a BB before it is first defined, do savings(v) = savings(v) + 1
- For every variable v computed in a BB, if it is live on exit from the BB, count a savings of 2 because it is not necessary to store it at the end of the BB
  - Assume load/store instructions cost two units
- Total savings per variable v is $\Sigma_{\text{blocks } B \text{ in } L} use(v, B) + 2 \times live(v, B)$
  - $live(v, B)$ is 1 or 0 depending on whether v is live on exit from B
- Estimate of savings is approximate since all BBs are not executed with the same frequency

Example of Global Register Allocation Based on Usage Counts



```
a = b * c
d = b - a
e = b / f
```

```
b = a - f
e = d + c
```

```
f = e * a
```

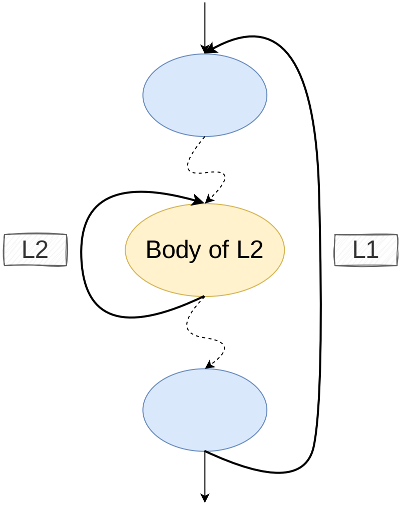
```
b = c - d
```

	B1	B2	B3	B4	Sum
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+0)	(1+0)	(0+0)	3
f	(1+0)	(1+0)	(0+2)	(0+0)	4

If there are 3 registers, they will be allocated to b, and (say) a and d

Global Register Allocation Based on Usage Counts for Nested Loops

- Assign registers for inner loops first before considering outer loops
- Let loop 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

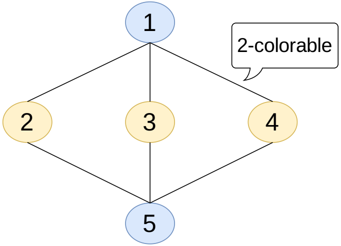


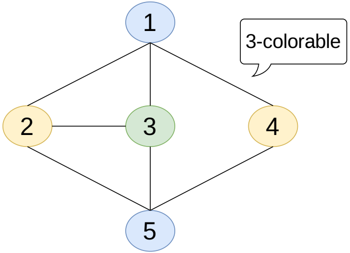
Global Register Allocation Using Graph Coloring

Chaitin’s allocator

The Graph Coloring Problem

- For an arbitrary graph G, a coloring of G assigns a color to each node in G so that no pair of adjacent nodes have the same color
- A coloring that uses k colors is termed a k-coloring
- The smallest possible k for a given graph is called the graph's chromatic number
- Determining if a graph is k-colorable, for some fixed k, is NP-complete





Global Register Allocation with Graph Coloring

An interference graph models conflicts in live regions

- If variables *a* and *b* are live at the same point, they cannot be assigned to the same register
- Nodes in an interference graph represent live ranges (LR) for a variable, and an edge (i,j) indicates LR_i and LR_j cannot share a register

Compilers model register allocation through *k*-coloring on an interference graph where *k* is the number of physical registers available to the allocator

- Each color represents an available register
- Spill some values to memory and retry if *k*-coloring is not possible

Identifying Global Live Ranges

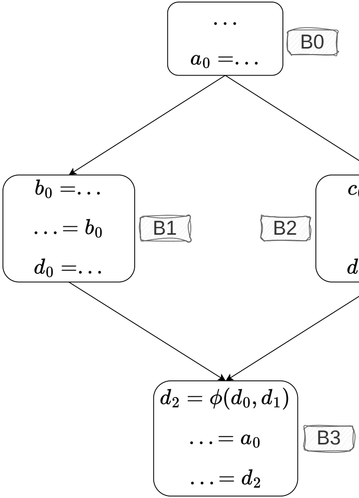
Requirement

- Group all definitions that reach a single use
- Group all uses that a single definition can reach

Assumptions

- Register allocation operates on the SSA form
  - In SSA, each name is defined once, and each use refers to one definition
  - ϕ functions are used at control flow merge points

Discovering Live Ranges Using SSA Form

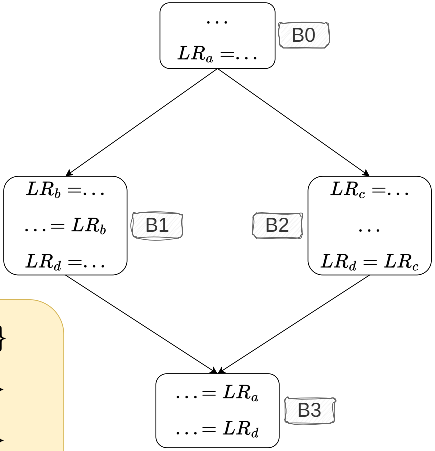


$$LR_a = \{a_0\}$$

$$LR_b = \{b_0\}$$

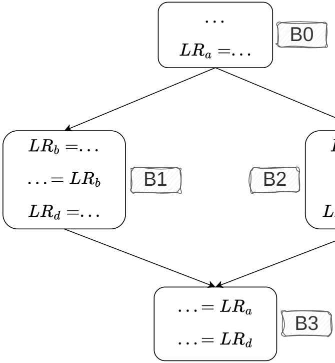
$$LR_c = \{c_0\}$$

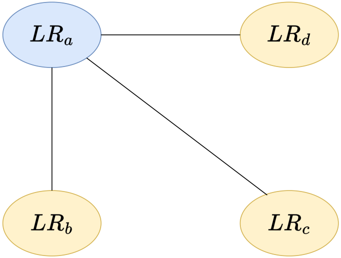
$$LR_d = \{d_0, d_1, d_2\}$$



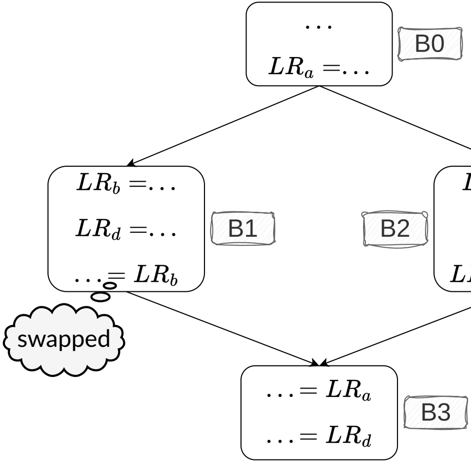
Interferences and the Interference Graph

- If there is an operation during which both LR_i and LR_j are live, they cannot reside in the same register
- Two live ranges LR_i and LR_j **interfere** if one is live at the definition of the other and they have different values





Interferences and the Interference Graph



B0

$\ldots$

$LR_a = \ldots$

B1

$LR_b = \ldots$

$LR_d = \ldots$

$\ldots = LR_b$

B2

$LR_c = \ldots$

$\ldots$

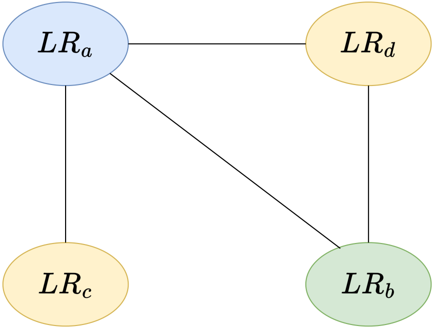
$LR_d = LR_c$

B3

$\ldots = LR_a$

$\ldots = LR_d$

swapped



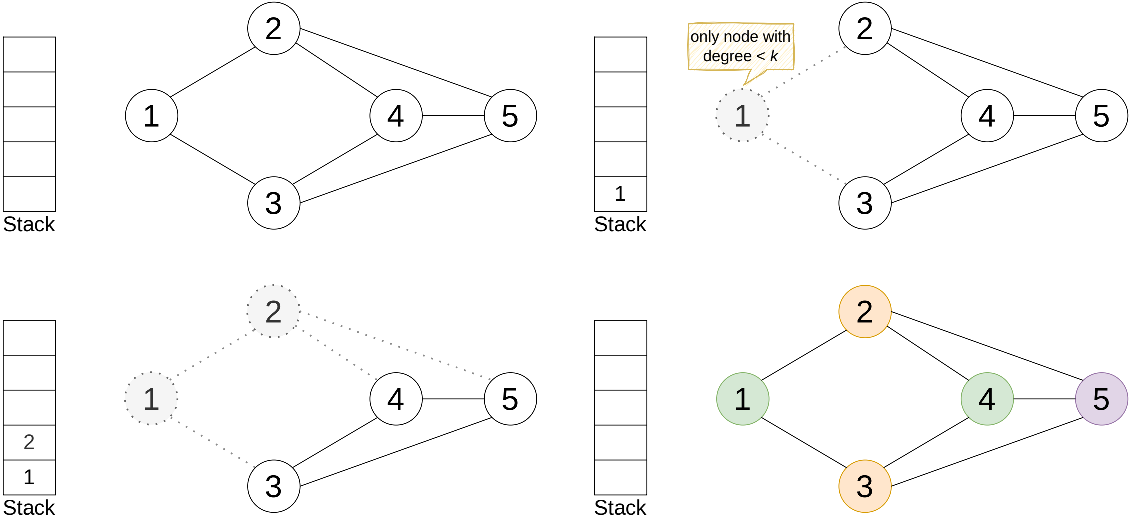
The allocator will need to spill either LR_a or LR_b if only two registers are available

Chaitin's Algorithm: High-level Idea

1. While ∃ vertices with degree less than *k*,
   - Choose an arbitrary node with degree less than *k* and put it on a stack
   - Remove that node and all its edges from the graph
   - This may decrease the degree of some other nodes and cause some more nodes to have a degree less than *k*
2. At some point, if all remaining vertices have a degree greater than or equal to *k*, then some node has to be spilled
   - Pick vertex *n* based on heuristic, spill live range of *n*
   - Remove vertex *n* and its incident edges from the graph, put *n* on "spill list"
   - Goto step 1
3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate
4. If no vertex needs to be spilled, successively pop vertices off the stack and color them in a color not used by neighbors (reuse colors as much as possible)

Chaitin’s Algorithm: High-Level Idea

Assume three physical registers are available



Detailed Steps in Chaitin's Algorithm

(i) Renaming — find all distinct live ranges and number them uniquely

(ii) Build — construct the interference graph

(iii) Coalesce — remove unnecessary copy or move operations for non-interfering variables

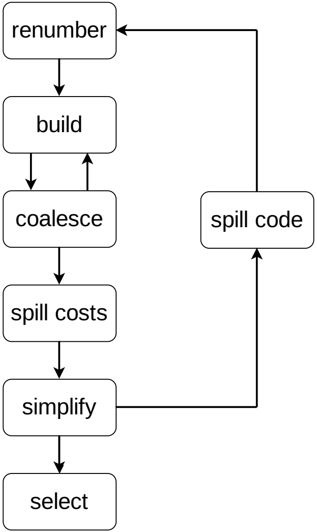
(iv) Spill costs — estimate the dynamic cost of spilling each live range

(v) Simplify — construct an ordering of the nodes in the interference graph

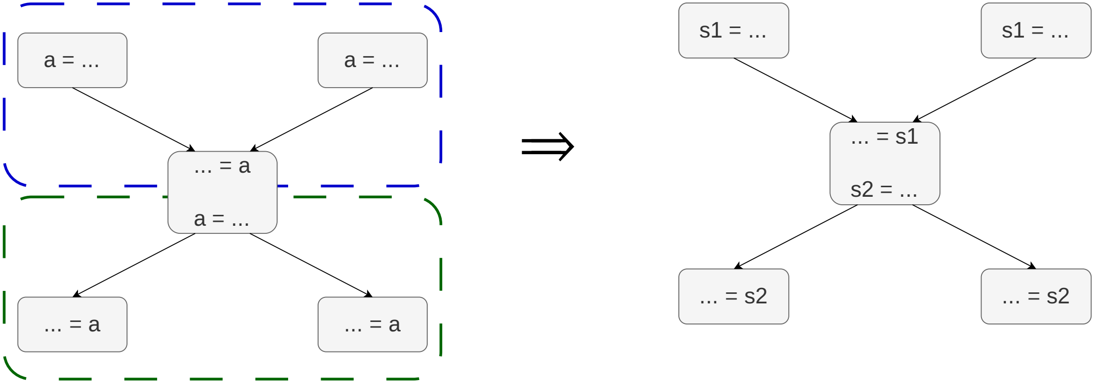
- Remove nodes with degree $< k$ from the graph and push them on a stack
- If every remaining node has degree $\geq k$, select a node, mark it for spilling, and remove it from the graph

(vi) Spill code — insert spill operations

(vii) Select — assign a register to each variable



Example of Renaming Live Ranges



Example of Renaming

Original Code

```
1. x = 2
2. y = 4
3. w = x + y
4. z = x + 1
5. u = x * y
6. x = z * 2
```

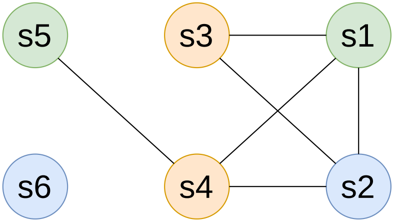
⇒

After Renaming

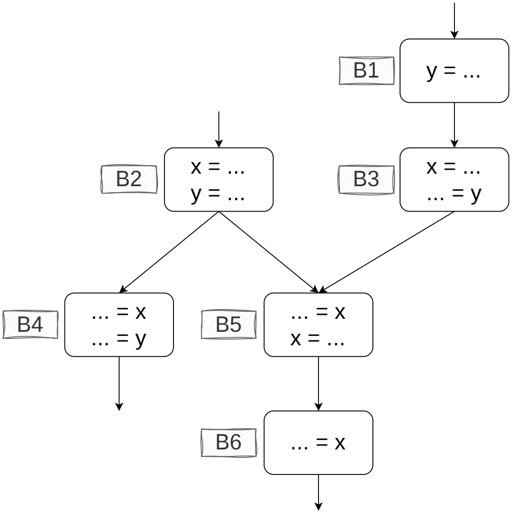
```
s1 = 2 // live range: 1-5
s2 = 4 // live range: 2-5
s3 = s1 + s2 // live range: 3-4
s4 = s1 + 1 // live range: 4-6
s5 = s1 * s2 // live range: 5-6
s6 = s4 * 2 // live range: 6-...
```

How to model constraints on register usage? For example, say s4 cannot live in register R1.

- Create nodes corresponding to the physical registers in the interference graph
- Add edges between interfering nodes (e.g., s4 and R1)



Live Range and Interference Graph

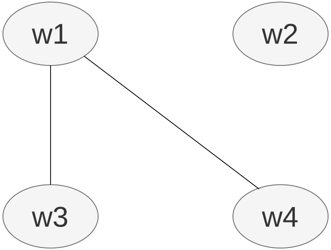


w1 def of x in B2, def of x in B3, use of x in B4, use of x in B5

w2 def of x in B5, use of x in B6

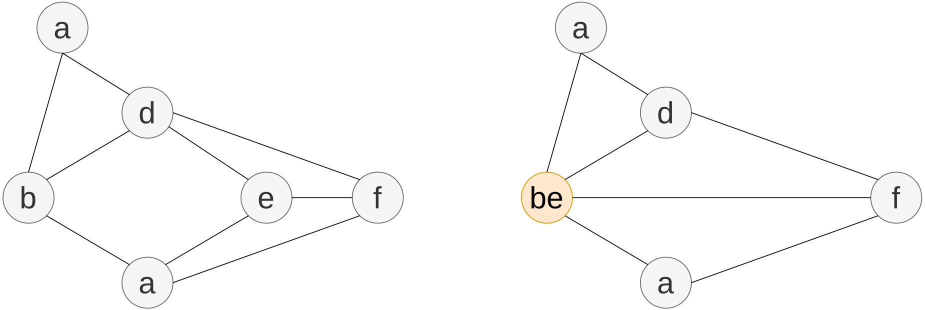
w3 def of y in B2, use of y in B4

w4 def of y in B1, use of y in B3

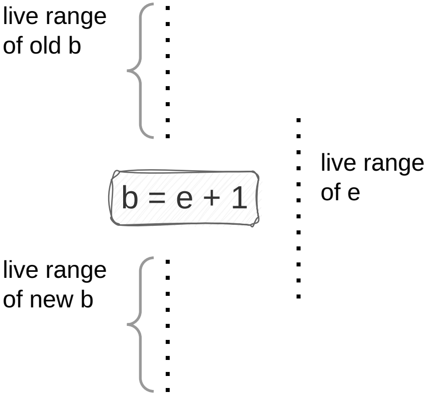


Coalescing

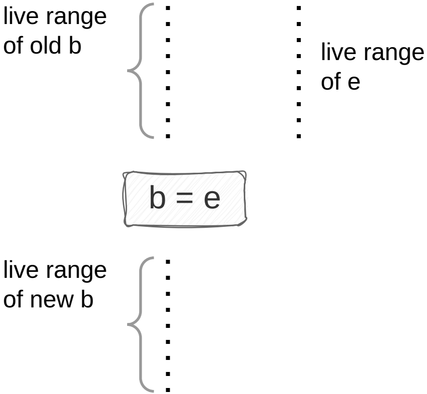
- Consider a copy instruction *b* = *e*
- If the live ranges of *b* and *e* do not overlap, then *b* and *e* can be given the same register (i.e., color)
  - The copy instruction can then be removed from the final program
- Coalesce by merging *b* and *e* into one node that contains edges of both nodes



Copy Subsumption or Coalescing

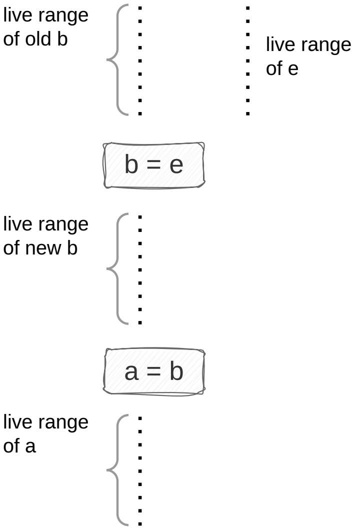


Copy subsumption is not applicable



Copy subsumption is possible, live ranges of e and new def of b do not interfere

Repeated Application of Copy Subsumption



Copy subsumption happens twice, once between b and e, and second between a and b.
e, b, and a can all be given the same register.

Coalescing

- Coalesce all possible copy instructions
- Rebuild the interference graph
  - May offer further opportunities for coalescing
  - Build-coalesce phase is repeated till no further coalescing is possible
- Coalescing reduces the size of the interference graph and possibly reduces spilling

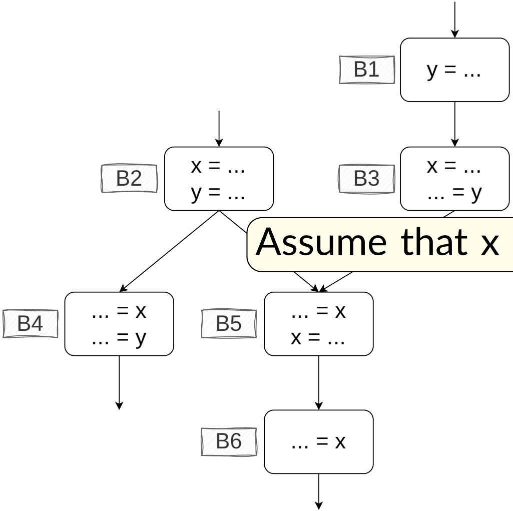
Splitting of Live Ranges

- Splitting live ranges creates an interference graph that is easier to color
  - Eliminates interference in a variable's "inactive" zones
  - Increases the flexibility of coloring, can now allocate the same variable to different registers

Estimating Spill Cost

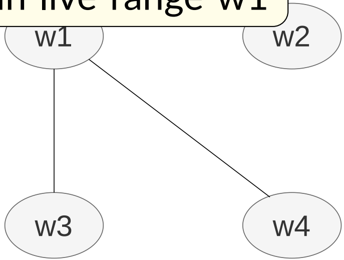
- If a node n in the interference graph has a degree less than k, remove n and all its edges from the graph and place n on a stack
- We need to spill a node when no such nodes are available
  - Implies loading a variable x into a register before every use of x and storing x from register into memory after every definition of x
- The higher the degree of a node to spill the greater the chance that it will help coloring
- Estimate of spill cost for a live range v: $cost(v) = \Sigma_{\text{all load or store operations in v}} c \times 10^d$, where c is the cost of the operation, d is the loop nesting depth, and 10 is the average number of loop iterations (assumption)
- The node to be spilled is the one with $\min(cost(v)/deg(v))$
- Negative spill costs are useful when there are only load and store operations to a memory location with no other uses
- Infinite spill cost is useful to model a definition followed immediately by a use

Effect of Spilling on Live Range

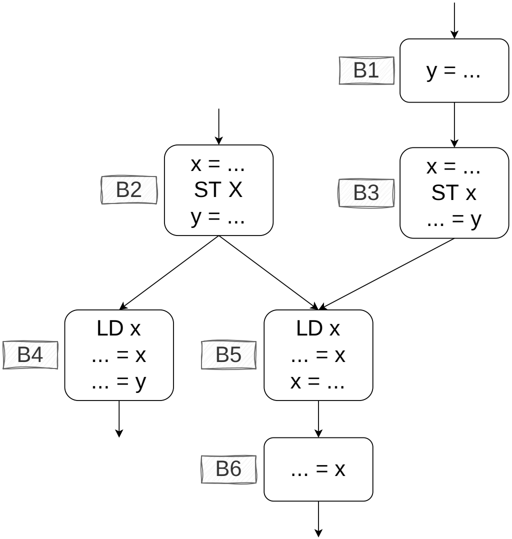


- w1 def of x in B2, def of x in B3, use of x in B4, use of x in B5
- w2 def of x in B5, use of x in B6
- w3 def of y in B2, use of y in B4
- w4 def of y in B1, use of y in B3

Assume that x is spilled in live range w1



Effect of Spilling on Live Range



w1 def of x in B2, ST of x in B2

w2 def of x in B3, ST of x in B3

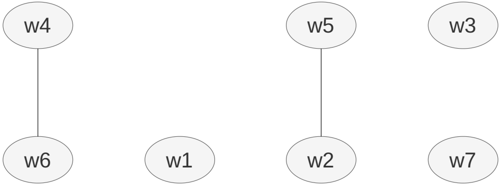
w3 def of x in B5, use of x in B6

w4 def of y in B2, use of y in B4

w5 def of y in B1, use of y in B3

w6 LD of x in B4, use of x in B4

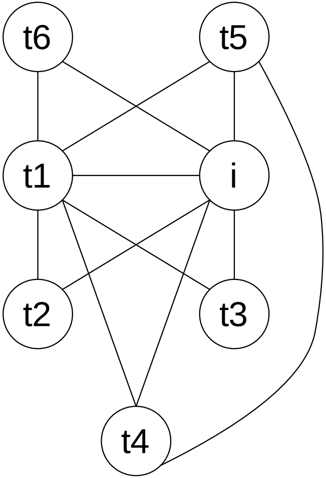
w7 LD of x in B5, use of x in B5



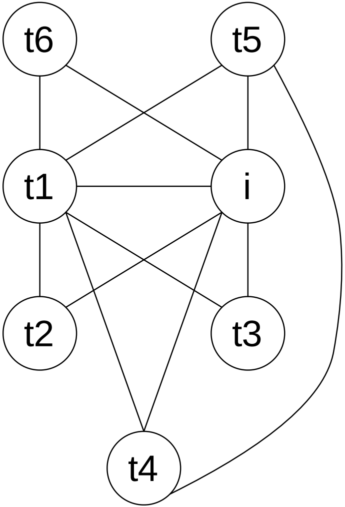
Chaitin's Algorithm: Detailed Example

```
 1      t1=202
 2      i=1
 3 L1:  t2=i>100
 4      if t2 goto L2
 5      t1=t1-2
 6      t3=addr(a)
 7      t4=t3-4
 8      t5=4*i
 9      t6=t4+t5
10      *t6=t1
11      i=i+1
12      goto L1
13 L2:
```

Variable	Live Range
t1	1–10
i	2–11
t2	3–4
t3	6–7
t4	7–9
t5	8–9
t6	9–10



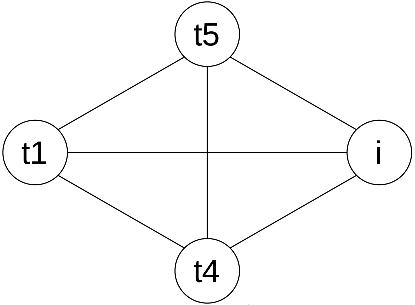
Chaitin's Algorithm: Detailed Example



⟹

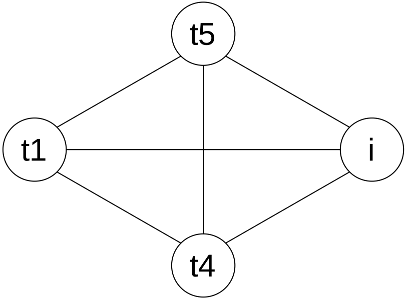
Assume there are three physical registers.

Nodes t6, t2, and t3 are pushed on to the stack during graph reduction.



cannot be reduced further, spilling is necessary

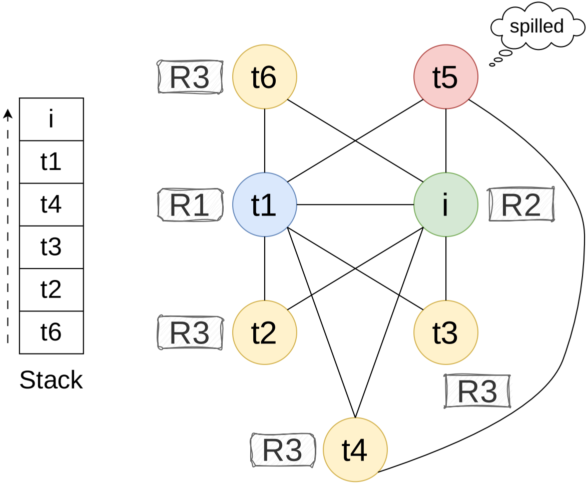
Chaitin's Algorithm: Detailed Example



Node	Cost(v)	deg(v)	Cost(v)/deg(v)
t1	1+(1+1+1)*10=31	3	10
i	1+(1+1+1+1)*10=41	3	14
t4	(1+1)*10=20	3	7
t5	(1+1)*10=20	3	7

Let us pick t5 for spilling

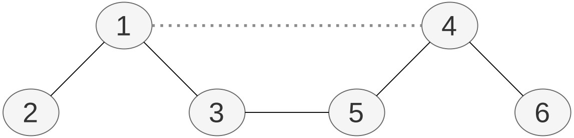
Chaitin's Algorithm: Detailed Example



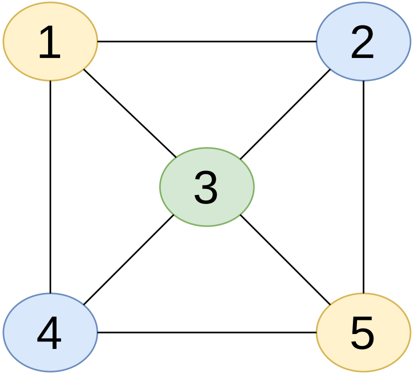
```
     R1=202
     R2=1
L1:  R3=R2>100
     if R3 goto L2
     R1=R1-2
     R3=addr(a)
     R3=R3-4
     t5=4*R2
     R3=R3+t5
     *R3=R1
     R2=R2+1
     goto L1
L2:
```

Problem with Chaitin's Algorithm

- Constructing and modifying the interference graphs are very costly
- Chaitin's original algorithm suggests always coalescing copy nodes



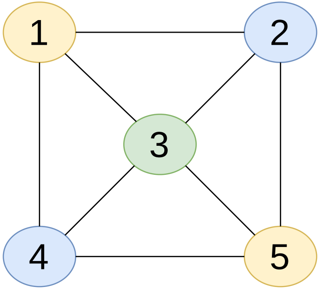
- Coalesce carefully
  - Do not coalesce if it increases the degree of a node to $\geq k$

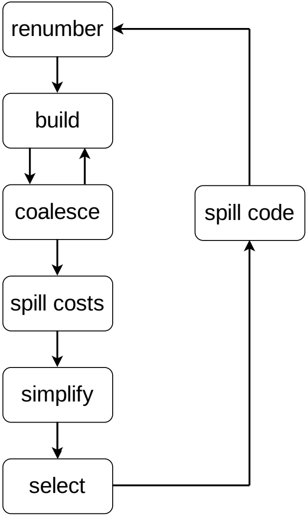


There is a 3-coloring, but Chaitin's heuristic does not find it

Briggs' Optimistic Allocator

- Instead of spilling, Briggs pushes the spill candidate on the stack, hoping there will be a color available
  - Spill a node only when it is popped and there are no colors available





Chaitin-Briggs' Algorithm: High-level Idea

1. While ∃ vertices with degree less than *k*,
    - Choose an arbitrary node with degree less than *k* and put it on a stack
    - Remove that node and all its edges from the graph
    - This may decrease the degree of some other nodes and cause some more nodes to have a degree less than *k*
2. At some point, if all remaining vertices have a degree greater than or equal to *k*, then some node may have to be spilled
    - Pick vertex *n* based on heuristic, do not spill
    - Remove vertex *n* and its incident edges from the graph, put *n* on the stack
    - Goto step 1
3. Successively pop vertices off the stack and color them using a color not used by some neighbor
    - If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it, and restart at step 1

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools. Section 8.8, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Chapter 13, 2nd edition, Morgan Kaufmann.
- Y. N. Srikanth. Global Register Allocation, NPTEL Course on Principles of Compiler Design.
- H. Leather. Compiler Optimization: Register Allocation