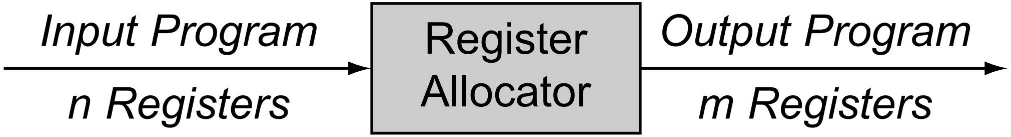
# 10 Register Allocation and Assignment

## Chapter Contents

1. Background Issues
2. Local Register Allocation and Assignment
    - Top-Down Local Register Allocator
    - Bottom-Up Local Register Allocator
3. Global Register Allocation and Assignment
    - Top-Down Global Register Allocator
    - Bottom-Up Global Register Allocator

# Register Allocation and Assignment

The register allocator takes a program that uses an arbitrary number of registers and produces an equivalent program that fits into the finite register set of the target machine.



Register allocators try to **minimize** the impact of the spill code that they must insert.

That impact can take at least **three** forms

- execution time for the spill code
- code space for the spill operations
- data space for the spilled values

Most allocators focus on the **first** of these effects.

# Memory versus Registers

The compiler writer's choice of a **memory model** defines many details of the allocation problem that the allocator must address.

## Register-to-register model

- knowledge about ambiguous memory references is encoded in the IR (*cf.* Slide 330)
- allocation is a necessary part of the process that produces legal code
- allocator inserts loads and stores to move some register-based values into memory

## Memory-to-memory model

- allocator must determine which values can be kept safely in registers
- allocator must determine whether keeping values in registers is profitable
- allocation improves performance by eliminating some loads and stores

The lack of information about ambiguity of values in the memory-to-memory model tends to favor the register-to-register model.

# Allocation versus Assignment

In a modern compiler, the register allocator solves two distinct but related problems.

**Allocation** maps an unlimited name space onto the register set of the target machine

- **register-to-register model**: virtual registers are mapped to name set modelling physical register set and values that do not fit in the register set are spilled
- **memory-to-memory model**: a subset of the memory locations is mapped to a set of names that models the physical register set

**Assignment** maps an allocated name set to the physical registers of the target machine

- assumes that allocation has been performed
- produces the actual register names required by the executable code

# Problem Complexity

Optimal allocation can be done in **polynomial time**

- for a single basic block,
- with one size of data value,
- if every value must be stored to memory at the end of its lifetime, and
- the cost of storing those values is uniform

Almost any additional complexity in the problem makes it **NP-complete**

- a second size of data item, *e.g.*, a register pair for a double precision number
- a memory model with nonuniform access costs
- if some values, *e.g.*, constants, need not be stored at the end of their lifetime
- extending the scope of allocation to include control flow and multiple blocks

In practice, one or more of these issues arise in compiling for any real system. In many cases, **all** of them arise.

# Register Classes

Most processors have distinct classes of registers for different kinds of values.

- **general-purpose registers** for integer values and memory locations
- **floating-point registers** to hold floating-point values
- separate registers for **condition codes**, *e.g.*, PowerPC
- **predicate registers** and **branch-target registers**, *e.g.*, Itanium (IA-64)

If interactions between two register classes are limited, the compiler may allocate them separately. If different register classes overlap, the compiler must allocate them together.

**Example** The common practice of using the same registers for single and double precision floating-point numbers forces the allocator to handle them as a single allocation problem.

# Local Register Allocation and Assignment

Local allocation addresses the problems that arise in producing a good allocation for a **single** basic block.

To simplify discussion, we assume the following

- the block the is the **entire program**: it loads the values that it needs from memory and it stores the values that it produces to memory
- the block uses a **single class** of general-purpose registers
- the target machine provides a **single set** of $k$ physical registers
- the code keeps any value that can legally reside in a register in a register, *i.e.*, it uses as many **virtual registers** as needed to encode this information
- the block contains a series of **three-address operations** $o_1, o_2, o_3, \ldots, o_N$ and each operation $o_i$ has the form $\texttt{op}_i\ \texttt{vr}_{i_1}\texttt{,}\ \texttt{vr}_{i_2} \Rightarrow \texttt{vr}_{i_3}$

The allocator must ensure that the output code **has no more than** $k$ values in registers at any point in the block.

# Top-Down Local Register Allocation

The **top-down local allocator** works from a simple heuristic: the **most heavily used** values should reside in registers.

- count the number of times (frequency) that each virtual register appears in the block
- allocates virtual registers to physical registers in descending order of frequency

If there are more virtual registers than physical registers, the allocator must reserve enough physical registers to load, store, and use the values that are not kept in registers.

$\mathcal{F}$

On any given ISA, $\mathcal{F}$ is the number of registers needed to generate code for values that live in memory. We pronounce $\mathcal{F}$ “feasible”.

# Top-Down Local Register Allocation

If the block uses **fewer than** `k` virtual registers, allocation is trivial and the compiler can simply assign each `vr` to its own physical register.

If the block uses **more than** `k` virtual registers, the compiler applies the following simple algorithm.

1. compute a priority for each virtual register
2. sort the virtual registers into priority order
3. assign the first $\mathtt{k} - \mathcal{F}$ virtual registers to physical registers in priority order
4. rewrite the code
    - replace virtual register names with physical register names
    - replace virtual register names with no allocated physical register with code that uses one of the reserved register and performs the appropriate load or store

# Top-Down Local Register Allocation

**Example** Assume a target machine with four physical registers $r_1$, $r_2$, $r_3$, and $r_4$ as well as two spill registers $f_1$ and $f_2$. Both $r_{arp}$ and $r_i$ are **live** upon entry into the block.

```
loadAI   r_arp, 12  ⇒  r_a
loadAI   r_arp, 16  ⇒  r_b
add      r_i, r_a   ⇒  r_c
sub      r_b, r_i   ⇒  r_d
mult     r_c, r_d   ⇒  r_e
multI    r_b, 2     ⇒  r_f
add      r_e, r_f   ⇒  r_g
storeAI  r_g        ⇒  r_arp, 8
```

| $r_{arp}$ | $r_a$ | $r_b$ | $r_c$ | $r_d$ | $r_e$ | $r_f$ | $r_g$ | $r_i$ |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 3 | 2 | 2 | 2 | 2 | 2 | 2 |

```
loadAI   r_arp, 12  ⇒  r_1
loadAI   r_arp, 16  ⇒  r_2
add      r_4, r_1   ⇒  r_3
sub      r_2, r_4   ⇒  f_1
storeAI  f_1        ⇒  r_arp, 20
mult     r_3, f_1   ⇒  f_1
storeAI  f_1        ⇒  r_arp, 24
multI    r_2, 2     ⇒  f_1
storeAI  f_1        ⇒  r_arp, 28
loadAI   r_arp, 24  ⇒  f_2
add      f_2, f_1   ⇒  f_1
storeAI  f_1        ⇒  r_arp, 8
```

# Bottom-Up Local Register Allocation

The primary weakness of top-down allocation is that it dedicates a physical register to one virtual register for the **entire** basic block.

The key idea behind the **bottom-up local allocator** is to focus on the details of how values are defined and used on an operation-by-operation basis.

1. start with an empty register set
2. allocate registers on demand
3. if no register is available, free one
    - keep values that will be “used soon” in registers
    - spill register whose next use is farthest in the future

```
foreach op$_i$ $vr_{i_1}$, $vr_{i_2}$ $\Rightarrow$ $vr_{i_3}$, in order
  $i = 1 \ldots N$ do
    $r_x \leftarrow$ Ensure($vr_{i_1}$, class($vr_{i_1}$))
    $r_y \leftarrow$ Ensure($vr_{i_2}$, class($vr_{i_2}$))
    if $vr_{i_1}$ is not needed after op$_i$ then
        Free($r_x$, class($r_x$))
    if $vr_{i_2}$ is not needed after op$_i$ then
        Free($r_y$, class($r_y$))
    $r_z \leftarrow$ Allocate($vr_{i_3}$, class($vr_{i_3}$))
    rewrite op$_i$ as op$_i$ $r_x$, $r_y$ $\Rightarrow$ $r_z$
    if $vr_{i_1}$ is needed after op$_i$ then
        class.Next[$r_x$] $\leftarrow$ Dist($vr_{i_1}$)
    if $vr_{i_2}$ is needed after op$_i$ then
        class.Next[$r_y$] $\leftarrow$ Dist($vr_{i_2}$)
    class.Next[$r_z$] $\leftarrow$ Dist($vr_{i_3}$)
```

The bottom-up allocator iterates over the operations in the block, making allocation decisions on demand.

By considering $vr_{i_1}$ and $vr_{i_2}$ in order, the allocator avoids using two physical registers for an operation with a repeated operand.

Trying to free $r_x$ and $r_y$ before allocating $r_z$ avoids spilling a register to hold an operation's result when the operation actually frees a register.

The function Dist(vr) returns the index in the block of the next reference to vr.

```
procedure Ensure(vr, class)
    if vr is already in class then
        result ← vr's physical register
    else
        result ← Allocate(vr, class)
        emit code to move vr into result
    return result
```

`Ensure` takes two arguments: a virtual register `vr` holding the desired value, and a representation for the appropriate register class `class`.

- if `vr` already occupies a physical register, `Ensure`'s job is done
- otherwise, it allocates a physical register for `vr` and emits code to move `vr`'s value into that physical register

In either case, it returns the physical register.

```
procedure Allocate(vr, class)
    if class.StackTop ≥ 0 then
        i ← pop(class)
    else
        i ← j that maximizes class.Next[j]
        store contents of j
    class.Name[i] ← vr
    class.Next[i] ← -1
    class.Free[i] ← false
    return i
```

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

`Allocate` returns a physical register from the free list of class, if one exists.

Otherwise, it selects the value stored in class that is used farthest in the future, spills it, and reallocates the corresponding physical register to `vr`.

```
procedure Free(vr, class)
    i ← push(class)
    class.Name[i] ← -1
    class.Next[i] ← ∞
    class.Free[i] ← true
```

`Free` simply pushes the freed register onto the stack and resets its fields to their initial values.

**Note** Some values in registers may not need to be stored on a spill

- known constant values
- values that were created by a load from memory

# Bottom-Up Local Register Allocation

**Example** Assume a target machine with four physical registers $r_1$, $r_2$, $r_3$, and $r_4$ as well as two spill registers $f_1$ and $f_2$. Both $r_{arp}$ and $r_i$ are **live** upon entry into the block.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| loadAI | $r_{arp}$, 12 | ⇒ | $r_a$ | loadAI | $r_{arp}$, 12 | ⇒ | $r_1$ |
| loadAI | $r_{arp}$, 16 | ⇒ | $r_b$ | loadAI | $r_{arp}$, 16 | ⇒ | $r_2$ |
| add | $r_i$, $r_a$ | ⇒ | $r_c$ | add | $r_4$, $r_1$ | ⇒ | $r_1$ |
| sub | $r_b$, $r_i$ | ⇒ | $r_d$ | sub | $r_2$, $r_4$ | ⇒ | $r_4$ |
| mult | $r_c$, $r_d$ | ⇒ | $r_e$ | mult | $r_1$, $r_4$ | ⇒ | $r_1$ |
| multI | $r_b$, 2 | ⇒ | $r_f$ | multI | $r_2$, 2 | ⇒ | $r_2$ |
| add | $r_e$, $r_f$ | ⇒ | $r_g$ | add | $r_1$, $r_2$ | ⇒ | $r_1$ |
| storeAI | $r_g$ | ⇒ | $r_{arp}$, 8 | storeAI | $r_1$ | ⇒ | $r_{arp}$, 8 |

**Note** Due to freeing operand registers before allocating result registers (*cf.* Lines 4–7 on Slide 516), the bottom-up allocator avoids all spills.

# Moving Beyond Single Blocks

We have seen how to build good allocators for single blocks

- working top down, we arrived at the frequency-count allocator
- working bottom up, we arrived at an allocator based on distance to the next use

However, local allocation does not capture the reuse of values across multiple blocks.

Moving from a single block to multiple blocks adds **many complications**.

- the allocator must correctly handle values computed in **previous** blocks
- the allocator must preserve values for use in **following** blocks

To accomplish this, the allocator needs a **more sophisticated way** of handling “values” than the local allocators use.

# Liveness and Live Ranges

Rather than allocating variables or values to registers, regional and global allocators compute a name space that is defined in terms of **live ranges**.

### Live range

A closed set of definitions and uses that are related to each other because their values flow together.

The term live range, implicitly, relies on the notion of **liveness**.

### Liveness

A variable $v$ is live at point $p$ if it has been defined along a path from the procedure's entry to $p$ and there exists a path from $p$ to a use of $v$ along which $v$ is not redefined.

# Liveness and Live Ranges

The set of live ranges is **distinct** from the set of variables and the set of values.

Every value computed in the code is part of some live range, even if it has **no name** in the original source code.

- intermediate results produced by address computations
- programmer-named variables
- array elements
- addresses loaded for use as branch targets

A single source-language variable may form **multiple** live ranges.

An allocator that works on live ranges can place **distinct** live ranges in **different** registers, *i.e.*, a source-language variable might reside in different registers at distinct points in the executing program.

# Liveness and Live Ranges

In straighline code, we can represent a live range as an interval $[i, j]$ where operation $i$ is its **definition** and operation $j$ is its **last use**.

```
1   loadI    ...        ⇒  r_arp
2   loadAI   r_arp, @a  ⇒  r_a
3   loadI    2          ⇒  r_2
4   loadAI   r_arp, @b  ⇒  r_b
5   loadAI   r_arp, @c  ⇒  r_c
6   loadAI   r_arp, @d  ⇒  r_x
7   mult     r_a, r_2   ⇒  r_a
8   mult     r_a, r_b   ⇒  r_a
9   mult     r_a, r_c   ⇒  r_a
10  mult     r_a, r_d   ⇒  r_a
11  storeAI  r_a        ⇒  r_arp, @a
```

| | Register | Interval |
|---|---|---|
| 1 | $r_{arp}$ | $[1, 11]$ |
| 2 | $r_a$ | $[2, 7]$ |
| 3 | $r_a$ | $[7, 8]$ |
| 4 | $r_a$ | $[8, 9]$ |
| 5 | $r_a$ | $[9, 10]$ |
| 6 | $r_a$ | $[10, 11]$ |
| 7 | $r_2$ | $[3, 7]$ |
| 8 | $r_b$ | $[4, 8]$ |
| 9 | $r_c$ | $[5, 9]$ |
| 10 | $r_d$ | $[6, 10]$ |

# Complications at Block Boundaries

To find live ranges in larger regions, the allocator must understand when a value is live **past the end** of the block that defines it.

At any point in the code, only live values need registers. LIVEOUT sets encode precisely this knowledge.

**LIVEOUT**

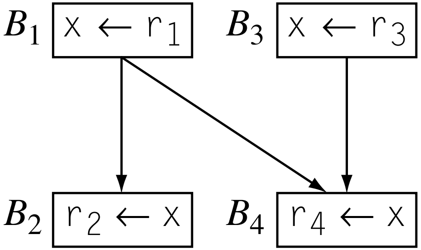
For each block $\mathtt{b}$, the set LIVEOUT$(\mathtt{b})$ contains all the variables that are live on exit from $\mathtt{b}$.

Any value in LIVEOUT$(\mathtt{b})$ must be **stored** to its assigned location in memory after its last definition in $\mathtt{b}$ to ensure that the correct value is available in a subsequent block.

# Complications at Block Boundaries

While LIVEOUT information allows the local allocator to produce correct code, that code will contain stores and loads to connect values **across** block boundaries.

**Example** The local allocator assigns the variable x to **different** registers in each block and uses store-load pair for correctness.



Along the edges $(B_1, B_2)$ and $(B_3, B_4)$, the compiler could replace the store-load pair with a register-to-register copy operation

- at the start of $B_2$ for $(B_1, B_2)$
- at the end of $B_3$ for $(B_3, B_4)$

However, edge $(B_1, B_4)$ **does not** have a location where the compiler can place the copy.

⟶ To avoid these problems, we need a **global** allocator that can coordinate the process of assigning registers across all the blocks.

# Global Register Allocation and Assignment

Global allocation differs from local allocation in **two** fundamental ways.

1. the structure of a global live range can be more complex than that of a local live range
2. the cost of spilling depends on where, *i.e.*, in which block, the spill code occurs

Each of these issues makes global allocation much more complex than local allocation.

We will look at **top-down allocators** that use high-level information and at **bottom-up allocators** that use low-level information to make allocation decisions.

First, we explore some of the subproblems that these allocators have in common

- discovering global live ranges
- estimating global spill costs
- building an interference graph

# Discovering Global Live Ranges

To construct live ranges, the compiler must discover the **relationships** that exist among different definitions and uses, *i.e.*, group together into a single name

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

The **SSA form** of the code (*cf.* Slide 265) provides a natural starting point

- each name is defined once, and each use refers to one definition
- $\phi$-function record the fact that distinct definitions on different paths in the control-flow graph reach a single reference

To build live ranges from SSA form, the allocator uses the **disjoint-set union-find algorithm** and makes a single pass over the code

1. treat each SSA name, or definition, as a set in the algorithm
2. union sets of each $\phi$-function parameter with the set for the $\phi$-function result

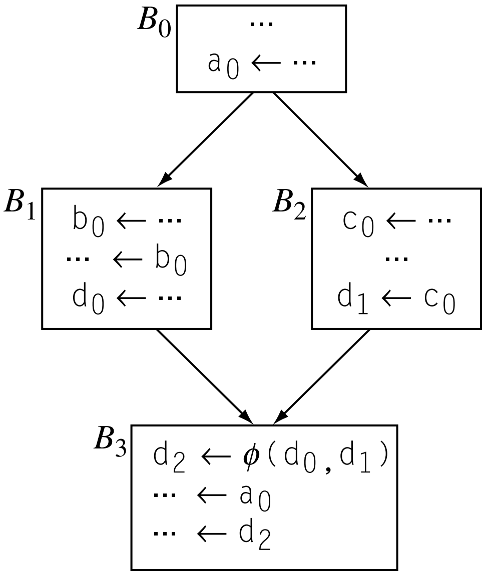
# Discovering Global Live Ranges

**Example** Consider the code fragment in SSA form shown on the right. It involves source-code variables a, b, c, and d.

To find the live ranges, the allocator assigns each ssa name a set containing its name. It unions together the sets associated with names used in the ϕ-function: $\{d_0\} \cup \{d_1\} \cup \{d_2\}$.

This gives a final set of four live ranges: $LR_a$ that contains $\{a_0\}$, $LR_b$ that contains $\{b_0\}$, $LR_c$ that contains $\{c_0\}$, and $LR_d$ that contains $\{d_0, d_1, d_2\}$.

At this point, the allocator can either **rewrite** the code to use live-range names or it can create and maintain a **mapping** between SSA names and live-range names.



# Estimating Global Spill Cost

The cost of a spill has three components: the **address computation**, the **memory operation**, and an **estimated execution frequency**.

## Address Computation

- to minimize the cost of the address computation spill into register-save area in AR
- use operations such as `loadAI` or `storeAI` relative to $\texttt{r}_{\texttt{arp}}$ for the spill

## Memory Operation

- the cost of the memory operation is, in general, unavoidable
- some embedded processors have scratchpad memory, *i.e.*, noncached local memory

## Estimated Execution Frequency

- profile data
- heuristics, *e.g.*, loops execute 10 times, `if-then-else` decreases frequency by half

# Estimating Global Spill Costs

The spill cost of a live range can also be **negative** or **infinite**.

Any live range with a negative spill cost should be spilled

- decreases demand for registers and removes instructions from the code
- *e.g.*, a live range that contains a load, a store, and no other uses

Some live ranges are so short that spilling them does not help

- assigning a spill cost of infinity ensures that the allocator does not try to spill it
- a live range should have infinite spill cost if no other live range ends between its definitions and its uses

# Interferences and the Interference Graph

The fundamental effect that a global register allocator must model is the **competition** among values for space in the processor's register set.

## Interference

Two live ranges, $LR_i$ and $LR_j$ interfere if one is live at the definition of the other and they have different values.

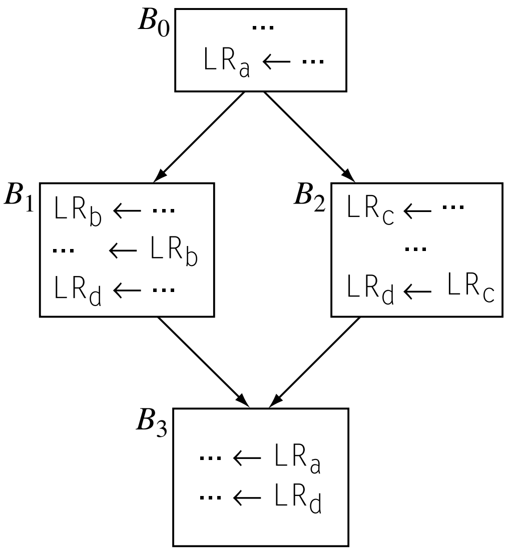
To model the allocation problem, the compiler can build an **interference graph** $I = (N, E)$

- nodes in $N$ represent individual live ranges
- edges in $E$ represent interferences between live ranges

An undirected edge $(n_i, n_j) \in I$ exists if and only if the corresponding live ranges $LR_i$ and $LR_j$ interfere.

# Interferences and the Interference Graph

**Example** On the left, the code (rewritten to use live ranges) from the example on Slide 529 is shown. The corresponding interference graph is shown on the right.

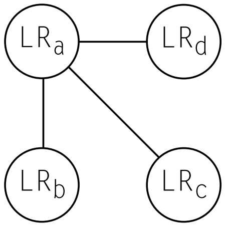


$B_0$: … $LR_a \leftarrow$ …

$B_1$: $LR_b \leftarrow$ … | … $\leftarrow LR_b$ | $LR_d \leftarrow$ …

$B_2$: $LR_c \leftarrow$ … | … | $LR_d \leftarrow LR_c$

$B_3$: … $\leftarrow LR_a$ | … $\leftarrow LR_d$



The interference graph shows that $LR_a$ interferes with **each** of the other live ranges.

The rest of the live ranges, however, **do not** interfere with each other.

# Interferences and the Interference Graph

Based on the interference graph the register allocation problem can be reformulated as a **graph-coloring problem**.

Graph coloring allocators attempt to construct a $k$-coloring for that graph, where $k$ is the number of physical registers available to the allocator.

- a $k$-coloring for the interference graph translates **directly** into an assignment of the live ranges to physical registers
- if the compiler cannot directly construct a $k$-coloring for the graph, it modifies the underlying code by spilling some values to memory and **tries again**

Because spilling simplifies the graph, this process is **guaranteed** to halt.

# Building the Interference Graph

Once the allocator has built global live ranges and annotated each block in the code with its LIVEOUT set, it can construct the interference graph in a linear pass over each block.

```
foreach $LR_i$ do
    create a node $n_i \in N$
foreach basic block $b$ do
    LIVENOW $\leftarrow$ LIVEOUT$(b)$
    foreach operation $op_n, op_{n-1} \ldots op_1$ in $b$ with form $op_i\ LR_a, LR_b \Rightarrow LR_c$ do
        foreach $LR_i \in$ LIVENOW do
            $E \leftarrow E \cup \{(LR_c, LR_i)\}$
        LIVENOW $\leftarrow$ LIVENOW $- LR_c$
        LIVENOW $\leftarrow$ LIVENOW $\cup \{LR_a\} \cup \{LR_b\}$
```

# Building the Interference Graph

The algorithm implements the definition of interference given earlier: $LR_i$ and $LR_j$ interfere only if one is live at a definition of the other.

⟶ A **copy** $LR_i \Rightarrow LR_j$ does not create an interference between $LR_i$ and $LR_j$ because the two live ranges have the same value and therefore can occupy the same register.

⟶ Likewise, a **$\phi$-function** does not create an interference between any of its arguments and its result.

**Note** To improve the allocator’s efficiency, the compiler can build both a **lower-diagonal bit matrix** and a set of **adjacency lists** to represent $E$.

- the bit matrix allows a constant-time test for interference
- the adjacency lists allows efficient iteration over a node’s neighbors

# Building an Allocator

To build a global allocator based on the graph-coloring paradigm, the compiler writer needs **two** additional mechanisms.

## An efficient technique to discover $k$-colorings

- determining if a $k$-coloring exists for a particular graph is NP-complete
- allocators use fast approximations that are not guaranteed to find a $k$-coloring

## A strategy that handles the case when no color remains for a specific live range

- the allocator picks one or more live ranges to modify
- **spilling** turns the chosen live range into sets of tiny live ranges, one at each definition or use of the original live range
- **splitting** breaks the chosen live range into smaller, but nontrivial, pieces

# Top-Down Coloring

Top-down allocators try to color live ranges in an order given by a **ranking function**.

The **priority-based**, **top-down allocators** assign each node a rank that is the estimated runtime savings that accrue from keeping that live range in a register.

1. consider live ranges in rank order and attempt to assign a color to each of them
2. if no color is available, invoke the spilling or splitting mechanism

To improve the process, the allocator can partition the live ranges into two sets: **constrained** live ranges ($k$ or more neighbors) and **unconstrained** live ranges.

- first, constrained live ranges are colored in rank order
- then, unconstrained live ranges are colored in any order

Because an unconstrained live range has fewer than $k$ neighbors, the allocator can always find a color for it: no assignment of colors to its neighbors can use all $k$ colors.

# Handling Spills

To spill $LR_i$, the allocator inserts a store after **every definition** of $LR_i$ and a load before **each use** of $LR_i$.

If memory operations need registers, the allocator can **reserve registers** to handle them.

- the number of registers needed is determined by the instruction set architecture
- reserving these registers simplifies spilling

An alternative to reserving registers is to **look for free colors** at each definition and use.

- if no color is available, retroactively spill a previously colored live range
- this scheme has the potential to spill previously colored live ranges recursively

This feature has led most implementors of top-down, priority-based allocators to reserve spill registers instead.

# Live-Range Splitting

Another way to change the problem is to **split an uncolored live range** into new live ranges, *i.e.*, subranges that contain several references.

- live-range splitting can avoid spilling the original live range at every reference
- well-chosen split points, it can isolate the portions of the live range that must be spilled

**Example** The first top-down, priority-based coloring allocator, broke the uncolored live range into single-block live ranges.

- it counted interferences for each resulting live range, and then recombined live ranges from adjacent blocks if the combined live range remained unconstrained
- it placed an arbitrary upper limit on how many blocks a split live range could span
- it inserted a load at the starting point of each split live range and a store at the live range’s ending point
- it spilled any split live ranges that remained uncolored

# Bottom-Up Coloring

The major distinction between top-down and bottom-up allocators lies in the **mechanism used to order** live ranges for coloring.

- a top-down allocator uses high-level information to select an order for coloring
- a bottom-up allocator computes an order from detailed structural knowledge about the interference graph

To order the live ranges, a bottom-up, graph-coloring allocator relies on the fact that unconstrained live ranges are **trivial** to color.

It assigns colors in an order where every node has **fewer** than $k$ colored neighbors.

# Bottom-Up Coloring

The algorithm computes the coloring order for a graph $I = (N, E)$ as follows.

```
initialize stack to empty
while $N \neq \emptyset$ do
    if $\exists n \in N : \mathrm{deg}(n) < k$ then
        node ← $n$
    else
        node ← $n$ picked from $N$
    remove node and its edges from $I$
    push node onto stack
```

It uses two **distinct mechanisms** to select the node to remove next.

- the first clause takes a node that is **unconstrained** in the graph from which it is removed
- the second clause, invoked only when every remaining node is **constrained**, picks a node using some external criteria

The loop halts when the graph is empty. At that point, the stack contains all the nodes in order of removal.

# Bottom-Up Coloring

To color the graph, the allocator rebuilds the interference graph in the order represented by the stack, *i.e.*, in **reverse order** of removing them from the graph.

```
while stack $\neq \emptyset$ do
    pop node from stack
    insert node and its edges into $\mathrm{I}$
    color node
```

The allocator tallies the colors of node's neighbors in the current approximation to $\mathrm{I}$ and assigns node an unused color.

If no color remains, node is left uncolored.

When the stack is empty, $\mathrm{I}$ has been rebuilt.

- if every node has a color, the allocator declares success and rewrites the code
- if nodes remain uncolored, the allocator spills or splits the corresponding live range

At this point, the classic bottom-up allocators rewrite the code to reflect the spills and splits and **repeat** the entire process.

# Coalescing Copies to Reduce Degree

The compiler writer can use the interference graph to determine when two live ranges that are connected by a copy can be **coalesced**, or combined.

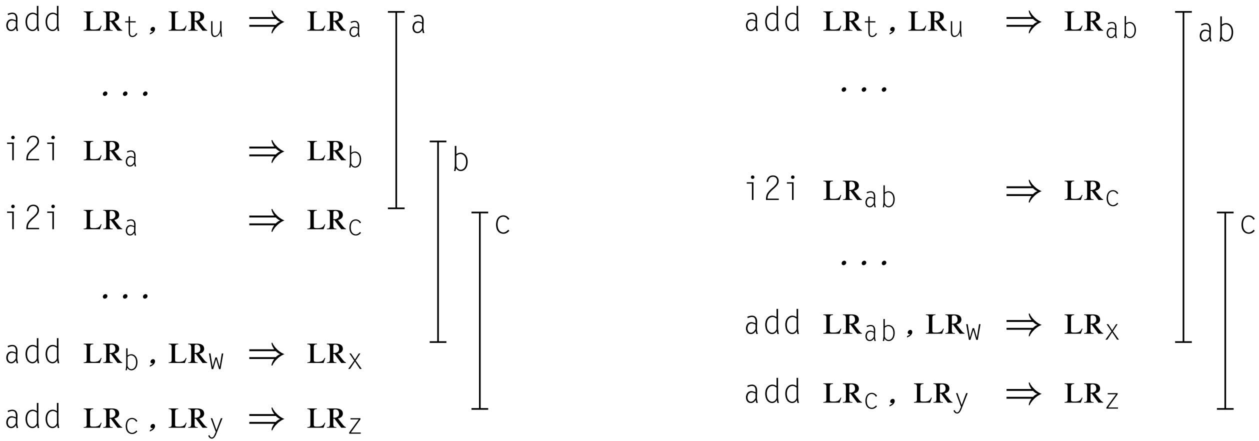
Combining two live ranges $LR_i$ and $LR_j$ has **several beneficial effects**

- it eliminates the copy operation, making the code smaller and, potentially, faster
- it reduces the degree of any $LR_k$ that interfered with both $LR_i$ and $LR_j$
- it shrinks the set of live ranges, making $\mathcal{I}$ and many related data structures smaller

Because these effects help in allocation, compilers often perform coalescing **before** the coloring stage in a global allocator.

# Coalescing Copies to Reduce Degree

**Example** The original code appears on the left, with lines to the right that indicate the regions where each of the relevant values, $LR_a$, $LR_b$, and $LR_b$ are live.



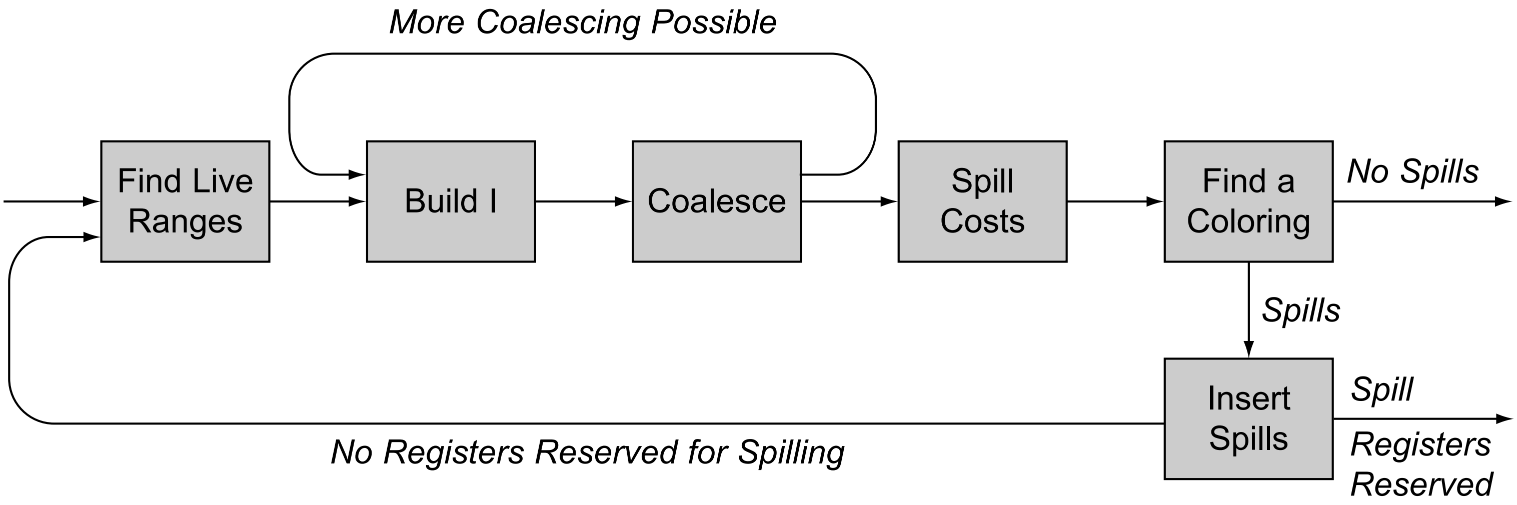
add $LR_t$ , $LR_u$ ⇒ $LR_a$
...
i2i $LR_a$ ⇒ $LR_b$
i2i $LR_a$ ⇒ $LR_c$
...
add $LR_b$ , $LR_w$ ⇒ $LR_x$
add $LR_c$ , $LR_y$ ⇒ $LR_z$

add $LR_t$ , $LR_u$ ⇒ $LR_{ab}$
...
i2i $LR_{ab}$ ⇒ $LR_c$
...
add $LR_{ab}$ , $LR_w$ ⇒ $LR_x$
add $LR_c$ , $LR_y$ ⇒ $LR_z$

On the right, the result of coalescing $LR_a$ and $LR_b$ to produce $LR_{ab}$ is shown. Since $LR_c$ is defined by a copy from $LR_{ab}$, they do not interfere.

# Comparing Top-Down and Bottom-Up Global Allocators

Both the top-down and the bottom-up coloring allocators have the **same basic structure**.


More Coalescing Possible
Find Live Ranges
Build I
Coalesce
Spill Costs
Find a Coloring
No Spills
Spills
Insert Spills
Spill
Registers Reserved
No Registers Reserved for Spilling