Chapter 13

Register Allocation

CHAPTER OVERVIEW

The code generated by a compiler must make effective use of the limited resources of the target processor. Among the most constrained resources is the set of hardware registers. Thus, most compilers include a pass that both allocates and assigns hardware registers to program values.

This chapter focuses on global register allocation and assignment via graph coloring; it describes the problems that occur at smaller scopes as a means of motivating a global allocator.

Keywords: Register Allocation, Register Spilling, Copy Coalescing, Graph-Coloring Allocators

13.1 INTRODUCTION

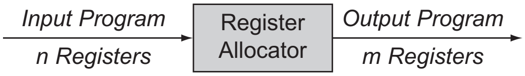
Registers are the fastest locations in the memory hierarchy. Often, they are the only memory locations that most operations can access directly. The proximity of registers to the functional units makes good use of registers a critical factor in runtime performance. In a compiler, responsibility for making good use of the target machine's register set lies with the register allocator.

The register allocator determines, at each point in the program, which values will reside in registers and which register will hold each of those values. If the allocator cannot keep a value in a register throughout its lifetime, the value must be stored in memory for some or all of its lifetime. The allocator might relegate a value to memory because the code contains more live values than the target machine's register set can hold. Alternatively, the value might be kept in memory between uses because the allocator cannot prove that it can safely reside in a register.

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00013-X

Conceptual Roadmap

Conceptually, the register allocator takes as its input a program that uses some arbitrary number of registers and produces. It takes as output an equivalent program that fits into the finite register set of the target machine.



The allocator may need to insert loads and stores to move values between registers and memory. The goal of register allocation is to make effective use of the target machine's register set and to minimize the number of loads and stores that the code must execute.

Register allocation plays a direct role in creating executable code that executes quickly, for the simple reason that register accesses are faster than memory accesses. At the same time, the algorithmic problems that underlie register allocation are hard—in their general form, they defy optimal solution. A good register allocator computes an effective approximate solution to a hard problem, and does it quickly.

Overview

To simplify the earlier parts of the compiler, most compilers use an IR in which the name space is not tied to either the address space of the target processor or its register set. To translate the IR code into assembly code for the target machine, these names must be mapped into the name space used in the target machine's ISA. Values stored in memory in the IR program must be turned into static coordinates that, in turn, map to runtime addresses using techniques such as those described in Section 6.4.3. Values stored in virtual registers in the IR must be mapped into the processors physical registers.

If the IR models computation with a memory-to-memory storage model, then the register allocator promotes memory-bound values into registers in the regions where they are heavily used. In this model, register allocation is an optimization that improves program performance by eliminating memory operations.

On the other hand, if the IR models the code with a register-to-register storage model, the register allocator must decide, at each point in the code, which virtual registers should reside in physical registers and which ones can live in memory. It constructs a map from virtual registers in the IR into some combination of physical registers and memory locations and rewrites the code to reflect that mapping. In this model, register allocation is required to create a correct target-machine program; it inserts loads and stores into the code and tries to place them where they will least hurt performance.

In general, the register allocator tries to minimize the impact of the loads and stores that it adds to the code, called *spill code*. That impact includes the time needed to execute the spill code, the code space that it occupies, and the data space occupied by the spilled values. A good register allocator tries to minimize all three.

Spill code
Loads and stores inserted by the register allocator are *spill code*.

The next section reviews some of the background issues that create the environment in which register allocators operate. Subsequent sections explore algorithms for register allocation and assignment in both local and global scopes.

13.2 BACKGROUND ISSUES

The register allocator takes as input code that is almost completely compiled—the code has been scanned, parsed, checked, analyzed, optimized, rewritten as target-machine code, and, perhaps, scheduled. The allocator must fit that code into the register set of the target machine by renaming values and inserting operations that move values between registers and memory. Many decisions made in earlier phases of the compiler affect the allocator's task, as do properties of the target machine's instruction set. This section explores several factors that play a role in shaping the role of the register allocator.

13.2.1 Memory versus Registers

The compiler writer's choice of a memory model defines many details of the allocation problem that the allocator must address (see Section 5.4.3). With a register-to-register model, earlier phases in the compiler directly encode their knowledge about ambiguous memory references into the shape of the IR; they place unambiguous values in virtual registers. Therefore, values stored in memory are assumed to be ambiguous (see Section 7.2), so the allocator leaves them in memory.

In a memory-to-memory model, the allocator does not have this code shape hint, because the IR program keeps all values in memory. In this model, the allocator must determine which values can be kept safely in registers—that is, which values are unambiguous. Next, it must determine whether keeping them in registers is profitable. In this model, the code that the allocator receives as input typically uses fewer registers and executes more memory operations than the equivalent register-to-register code. To obtain good performance, the allocator must promote as many of the memory-based values into registers as it can.

Thus, the choice of memory model fundamentally determines the allocator's task. In both scenarios, the allocator's goal is to reduce the number of loads and stores that the final code executes to move values back and forth between registers and memory. In a register-to-register model, allocation is a necessary part of the process that produces legal code; it ensures that the final code fits into the target machine's register set. The allocator inserts load and store operations to move some register-based values into memory—presumably in regions where demand for registers exceeds supply. The allocator tries to minimize the impact of the load and store operations that it inserts.

In contrast, in a compiler with a memory-to-memory model, the compiler performs register allocation as an optimization. The code is legal before allocation; allocation merely improves performance by keeping some memory-based values in registers and eliminating the loads and stores used to access them. The allocator tries to remove as many loads and stores as possible, since this can significantly improve the final code's performance.

Thus, lack of knowledge—limitations in the compiler's analysis—may keep the compiler from allocating a variable to a register. It can also occur when a single code sequence inherits different environments along different paths. These limitations on what the compiler may know tend to favor the register-to-register model. The register-to-register model provides a mechanism for other parts of the compiler to encode knowledge about ambiguity and uniqueness. This knowledge might come from analysis, it might come from understanding the translation of a complex construct, or it might even be derived from the source text in the parser.

13.2.2 Allocation versus Assignment

In a modern compiler, the register allocator solves two distinct problems—register allocation and register assignment—that have sometimes been handled separately in the past. These problems are related but distinct.

1. *Allocation* Register allocation maps an unlimited name space onto the register set of the target machine. In a register-to-register model, register allocation maps virtual registers to a new set of names that models the physical register set and spills values that do not fit in the register set. In a memory-to-memory model, it maps some subset of the memory locations to a set of names that models the physical register set. Allocation ensures that the code will fit the target machine's register set at each instruction.
2. *Assignment* Register assignment maps an allocated name set to the physical registers of the target machine. Register assignment assumes that allocation has been performed, so the code will fit into the set of physical registers provided by the target machine. Thus, at each

instruction in the generated code, no more than k values are designated as residing in registers, where k is the number of physical registers. Assignment produces the actual register names required by the executable code.

Register allocation is a hard problem. General formulations of the problem are NP-complete. For a single basic block, with one size of data value, optimal allocation can be done in polynomial time, 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. For example, adding a second size of data item, such as a register pair that holds a double-precision floating-point number, makes the problem NP-complete. Alternately, adding a memory model with nonuniform access costs, or the distinction that some values, such as constants, need not be stored at the end of their lifetime, makes the problem NP-complete. Extending the scope of allocation to include control flow and multiple blocks also makes the problem NP-complete. In practice, one or more of these issues arise in compiling for any real system. In many cases, all of them arise.

Register assignment, in many cases, can be solved in polynomial time. Assume a machine with one kind of register. Given a feasible allocation for a basic block—that is, one in which the demand for physical registers at each instruction does not exceed the number of physical registers—an assignment can be produced in linear time using an analog of *interval-graph* coloring. The related problem for an entire procedure can be solved in polynomial time—that is, if, at each instruction, the demand for physical registers does not exceed the number of physical registers, then the compiler can construct an assignment in polynomial time.

Interval graph
An *interval graph* represents the overlap between multiple intervals on the real line. It has a node for each interval and an edge (i,j) if and only if i and j have a non-empty intersection.

The distinction between allocation and assignment is both subtle and important. In seeking to improve a register allocator's performance, the compiler writer must understand whether the weakness lies in allocation or assignment and direct effort to the appropriate part of the algorithm.

13.2.3 **Register Classes**

The physical registers provided by most processors do not form a homogenous pool of interchangeable resources. Most processors have distinct classes of registers for different kinds of values.

For example, most modern computers have both *general-purpose registers* and *floating-point registers*. The former hold integer values and memory addresses, while the latter hold floating-point values. This dichotomy is not new; the early IBM 360 machines had 16 general-purpose registers and four floating-point registers. Modern processors may add more classes.

For example, the PowerPC has a separate register class for condition codes, and the Intel IA-64 has additional classes for predicate registers and branch-target registers. The compiler must place each value in a register of the appropriate class.

If the interactions between two register classes are limited, the compiler may allocate registers for them independently. On most processors, general-purpose registers and floating-point registers are not used to hold the same kinds of values. Thus, the compiler can allocate the floating-point registers independently from the general-purpose registers. The fact that the compiler uses general-purpose registers to spill floating-point registers means that it should allocate the floating-point registers first. Breaking allocation into smaller problems in this way reduces the size of the data structures, and may produce faster compile times.

The values in floating-point registers have a different source-language type, so they are disjoint from the values stored in general-purpose registers.

If, on the other hand, different register classes overlap, the compiler must allocate them together. 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—whether a double-precision value uses two single-precision registers or a single-precision value uses one-half of a double-precision register. A similar problem arises on architectures that allow values of different length to be stored in general-purpose registers. For example, the ISAs derived from the Intel x86 allow some 32-bit registers to hold one 32-bit value, two 16-bit values, or four 8-bit values. The allocator must model both potential uses and conflicts between those uses.

13.3 LOCAL REGISTER ALLOCATION AND ASSIGNMENT

As an introduction to register allocation, consider the problems that arise in producing a good allocation for a single basic block—local allocation, to use the terminology from optimization (see Section 8.3). A local allocator operates on one block.

To simplify the discussion, we assume that the block is the entire program. It loads the values that it needs from memory. It stores the values that it produces to memory. The input block uses a single class of general-purpose registers; the techniques extend easily to handle multiple disjoint register classes. The target machine provides a single set of *k* physical registers.

The code shape encodes information about which values can legally reside in a register for nontrivial amounts of time. The code keeps any value that can legally reside in a register in a register. It uses as many virtual registers as needed to encode this information; thus, the input block may name more than *k* virtual registers.

The input block contains a series of three-address operations $\mathtt{o}_1$, $\mathtt{o}_2$, $\mathtt{o}_3$, …, $\mathtt{o}_N$. Each operation, $\mathtt{o}_i$, has the form $\mathtt{op}_i\ \mathtt{vr}_{i_1}, \mathtt{vr}_{i_2} \Rightarrow \mathtt{vr}_{i_3}$. From a high-level view, the goal of local register allocation is to create an equivalent block in which each reference to a virtual register is replaced with a reference to a specific physical register. If the number of virtual registers is greater than *k*, the allocator may need to insert loads and stores to fit the code into the *k* physical registers. An alternative statement of this property is that the output code can have no more than *k* values in registers at any point in the block.

We use $\mathtt{vr}_i$ to denote a virtual register and $\mathtt{r}_i$ to denote a physical register.

This section explores two approaches to local allocation. The first approach counts the number of references to a value in the block and uses these “frequency counts” to determine which values reside in registers. Because it relies on externally derived information—the frequency counts—to prioritize the allocation of virtual to physical registers, we consider this a top-down approach. The second approach relies on detailed, low-level knowledge of the code to make its decisions. It walks over the block and determines, at each operation, whether or not a spill is needed. Because it synthesizes and combines many low-level facts to drive its decision-making process, we consider this a bottom-up approach.

13.3.1 **Top-Down Local Register Allocation**

The top-down local allocator works from a simple principle: the most heavily used values should reside in registers. To implement this heuristic, it finds the number of times that each virtual register appears in the block. Then, it allocates virtual registers to physical registers in descending order by frequency count.

If there are more virtual registers than physical registers, the allocator must reserve enough physical registers to allow it to load, store, and use the values that are not kept in registers. The precise number of registers that it needs depends on the processor. A typical RISC machine might need two to four registers. We will refer to this machine-specific number as $\mathcal{F}$.

$\mathcal{F}$

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."

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. In this case, the allocator does not need to set aside the $\mathcal{F}$ physical registers for spill code. If the block uses more than *k* virtual registers, the compiler applies the following simple algorithm:

1. *Compute a priority for each virtual register* In a linear pass over the operations in the block, the allocator tallies the number of times each

virtual register appears. This frequency count is the virtual register's priority.

2. *Sort the virtual registers into priority order* Priorities vary between two and the block length, so the best sorting algorithm depends on block length.
3. *Assign registers in priority order* Assign the first $k - \mathcal{F}$ virtual registers to physical registers.
4. *Rewrite the code* In a linear pass over the code, the allocator rewrites the code. It replaces virtual register names with physical register names. Any reference to a virtual register name with no allocated physical register is replaced with a short sequence that uses one of the reserved register and performs the appropriate load or store operation.

Top-down local allocation keeps heavily used virtual registers in physical registers. Its primary weakness lies in its approach to allocation—it dedicates a physical register to one virtual register for the entire basic block. Thus, a value that sees heavy use in the first half of the block and no use in the second half of the block effectively wastes that register through the second half of the block. The next section presents a technique that addresses this problem. It takes a fundamentally different approach to allocation—a bottom-up, incremental approach.

13.3.2 Bottom-Up Local Register Allocation

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. The bottom-up local allocator begins with all the registers unoccupied. For each operation, the allocator needs to ensure that its operands are in registers before it executes. It must also allocate a register for the operation's result. Figure 13.1 shows its basic algorithm, along with three support routines that it uses.

The bottom-up allocator iterates over the operations in the block, making allocation decisions on demand. There are, however, some subtleties. By considering $\texttt{vr}_{\texttt{i}_1}$ and $\texttt{vr}_{\texttt{i}_2}$ in order, the allocator avoids using two physical registers for an operation with a repeated operand, such as add $r_y, r_y \Rightarrow r_z$. Similarly, 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. Most of the complications in the algorithm occur in the routines *Ensure*, *Allocate*, and *Free*.

The routine *Ensure* is conceptually simple. It takes two arguments, a virtual register, *vr*, holding the desired value, and a representation for the

```
/* the bottom-up local allocator */
for each operation, i, in order from 1
   to N where i has the form
      op $vr_{i_1}$ $vr_{i_2}$ ⇒ $vr_{i_3}$
   $r_x$ ← Ensure($vr_{i_1}$ , class($vr_{i_1}$ ))
   $r_y$ ← Ensure($vr_{i_2}$ , class($vr_{i_2}$ ))
   if $vr_{i_1}$ is not needed after i
     then Free($r_x$ , class($r_x$ ))
   if $vr_{i_2}$ is not needed after i
     then Free($r_y$ , class($r_y$ ))
   $r_z$ ← Allocate($vr_{i_3}$ , class($vr_{i_3}$ ))
   rewrite i as $op_i$ $r_x$ , $r_y$ ⇒ $r_z$
   if $vr_{i_1}$ is needed after i
     then class.Next[$r_x$ ] ← Dist($vr_{i_1}$ )
   if $vr_{i_2}$ is needed after i
     then class.Next[$r_y$ ] ← Dist($vr_{i_2}$ )
   class.Next[$r_z$ ] ← Dist($vr_{i_3}$ )
```

```
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

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
```

■ **FIGURE 13.1** The Bottom-Up, Local Register Allocator.

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.

`Allocate` and `Free` expose the details of the allocation problem. To understand them, we need a concrete representation for a register class, shown in the C code to the left. A class has `Size` physical registers, each of which is represented by a virtual register name (`Name`), an integer that indicates the distance to its next use (`Next`), and a flag indicating whether or not that physical register is currently in use (`Free`). To initialize the class structure, the compiler sets each register to an unallocated state (say, `class.Name` as an invalid name, `Class.Next` as ∞, and `class.Free` as true), and pushes each of them onto the class' stack.

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

At this level of detail, both `Allocate` and `Free` are straightforward. Each class has a stack of free physical registers. `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`. `Allocate` sets the `Next` field to −1 to ensure that this register is not chosen for the other operand in the current operation. The allocator resets this field after it finishes with the current operation. `Free` simply needs to push the freed register onto the stack and reset its fields to their initial values. The function `Dist(vr)` returns the index in the block of the next reference to `vr`. The compiler can precompute this information in a backward pass over the block.

The bottom-up local allocator operates in an intuitive way. It assumes that the physical registers are initially empty and it places them all on a free list. It satisfies demand for registers from the free list, until that list is exhausted. After that, it satisfies demand by spilling some value to memory and reusing that value's register. It always spills the value whose next use is farthest in the future. Intuitively, it selects the register that would otherwise be unreferenced for the longest period of time. In some sense, it maximizes the benefit obtained for the cost of the spill.

In practice, this algorithm produces excellent local allocations. Indeed, several authors have argued that it produces optimal allocations. However, complications arise that cause it to produce suboptimal allocations. At any point in the allocation, some values in registers may need to be stored on a spill, while others may not. For example, if the register contains a known constant value, the store is superfluous since the allocator can recreate the value without a copy in memory. Similarly, a value that was created by a load from memory need not be stored. A value that need not be stored is called *clean*, while a value that needs a store is called *dirty*.

To produce an optimal local allocation, the allocator must take into account the difference in cost between spilling clean values and spilling dirty values. Consider, for example, allocation on a two-register machine, where the values x_1 and x_2 are already in the registers. Assume that x_1 is clean and x_2 is dirty. If the reference string for the remainder of the block is x_3 x_1 x_2, the allocator must spill one of x_1 or x_2. Since x_2's next use lies farthest in the future, the bottom-up local algorithm would spill it, producing the sequence of memory operations shown on the left. If, instead, the allocator spills x_1, it produces the shorter sequence of memory operations shown on the right.

store x_2
load x_3
load x_2

Spill Dirty Value

load x_3 (overwriting x_1)
load x_1

Spill Clean Value

This scenario suggests that the allocator should preferentially spill clean values over dirty values. The answer is not that simple.

Consider another reference string, $x_3\ x_1\ x_3\ x_1\ x_2$, with the same initial conditions. Consistently spilling the clean value produces the sequence of four memory operations on the left. In contrast, consistently spilling the dirty value produces the sequence on the right, which requires fewer memory operations.

```
   load x3
   load x1                  store x2
   load x3                  load  x3
   load x1                  load  x2
Spill Clean Value       Spill Dirty Value
```

Thepresence of both clean and dirty values makes optimal local allocation NP-hard. Still, the bottom-up local allocator produces good local allocations in practice. The allocations tend to be better than those produced by the top-down algorithm.

In local allocation, "optimal" means the allocation with the fewest spills.

13.3.3 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. Because such reuse occurs routinely, we need allocators that extend their scope across multiple blocks.

Unfortunately, moving from a single block to multiple blocks adds many complications. For example, our local allocators assumed implicitly that values do not flow between blocks. The primary reason for moving to a larger scope for allocation is to account for the flow of values between blocks and to generate allocations that handle such flows efficiently. The allocator must correctly handle values computed in previous blocks, and it 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

Regional and global allocators try to assign values to registers in a way that coordinates their use across multiple blocks. We saw, in both the top-down allocator and in the earlier discussion of SSA form (see Section 9.3), that the compiler can sometimes compute a new name space that better serves the purposes of a given algorithm. Regional and global allocators rely on this observation; they compute a name space that reflectsthe actual patterns of definitions and uses for each value. Rather than allocating variables or values to registers, these allocators compute a name space that is defined in terms of *live ranges*.

Live range
a closed set of related definitions and uses that serves as the base name space for register allocation

A single live range consists of a set of definitions and uses that are related to each other because their values flow together. That is, a live range contains a set of definitions and a set of uses. This set is self-contained in the sense that, for each use, every definition that can reach that use is in the same live range as the use. Similarly, for each definition, every use that can refer to the result of the definition is in the same live range as the definition.

The term *live range* relies, implicitly, on the notion of *liveness*, as described in Section 8.6.1. Recall that 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. Anywhere that *v* is live, its value must be preserved because subsequent execution might use *v*. Remember, *v* can be either a source-program variable or a compiler-generated temporary.

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. Thus, the intermediate results produced by address computations are live ranges, as do programmer-named variables, array elements, and 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. Thus, a source-language variable might reside in different registers at distinct points in the executing program.

To make these ideas concrete, first consider the problem of finding live ranges in a single basic block. Figure 13.2 repeats the ILOC code that we first encountered in Figure 1.3, with the addition of an initial operation that defines r_{arp}. The table on the right side shows the distinct live ranges in the block. In straightline code, we can represent a live range as an interval. Notice that each operation defines a value and, thus, starts a live range. Consider r_{arp}. It is defined in operation 1. Every other reference to r_{arp} is a use. Thus, the block uses just one value for r_{arp}, which is live over the interval [1,11].

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

For live ranges that span multiple blocks, we need a more complex notation.

In contrast, r_a has several live ranges. Operation 2 defines it; operation 7 uses the value from operation 2. Operations 7, 8, 9, and 10 each define a new value for r_a; in each case, the following operation uses the value. Thus,

```
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]

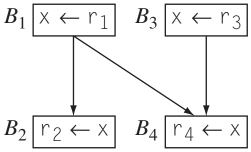
■ **FIGURE 13.2** Live Ranges in a Basic Block.

the value named r_a in the original code corresponds to five distinct live ranges: [2,7], [7,8], [8,9], [9,10], and [10,11]. A register allocator need not keep these distinct live ranges in the same physical register. Instead, it can treat each live range in the block as an independent value for allocation and assignment.

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. LiveOut sets, as computed in Section 8.6.1, encode precisely this knowledge. At any point in the code, only live values need registers. Thus, LiveOut sets play a key role in register allocation.

Complications at Block Boundaries

A compiler that uses local register allocation might compute LiveOut sets for each block as a necessary prelude to provide the local allocator with information about the status of values at the block's entry and exit. LiveOut sets allows the allocator to handle the end-of-block conditions correctly. Any value in LiveOut(*b*) must be stored to its assigned location in memory after its last definition in *b* to ensure that the correct value is available in a subsequent block. In contrast, a value that is not in LiveOut(*b*) can be discarded without a store after its last use in *b*.

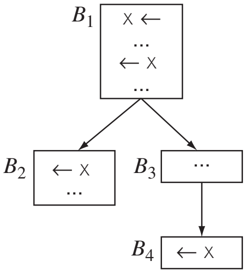


While LiveOut information allows the local allocator to produce correct code, that code will contain stores and loads whose sole purpose is to connect values across block boundaries. Consider the example shown in the margin. The local allocator has assigned the variable x to different registers in each block: r_1 in B_1, r_2 in B_2, r_3 in B_3, and r_4 in B_4. The only local mechanism to resolve these conflicting assignments is to store x at the end of B_1 and B_3 and to load it at the start of B_2 and B_4, as shown. This solution passes the value of x through memory to move it into its assigned register in B_2 and B_4.

Along the control-flow 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 in the appropriate place: the start of B_2 for (B_1, B_2) and 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 because it is a critical edge, as discussed in Section 9.3.5. Placing the copy at the end of B_1 produces an incorrect assignment for B_2, while placing it at the start of B_4 produces an incorrect result on edge (B_3, B_4).

The local allocator cannot, in general, use copy operations to connect the flow of values between blocks. It cannot know, when processing B_1, the allocation and assignment decisions made in subsequent blocks. Thus, it must resort to passing the values through memory. Even if the allocator knew the assignments in B_2 and B_4 when it processed B_1, it still cannot resolve the problem with (B_1, B_4) unless it changes the control-flow graph. Alternately, the allocator could avoid these problems by coordinating the assignment process across all the blocks. At that point, however, the allocator would no longer be a local allocator.

Similar effects arise with allocation. What if x were not referenced in B_2? Even if we could coordinate assignment globally, to ensure that x was always in some register, say r_2, when it was used, the allocator would need to insert a load of x at the end of B_2 to let B_4 avoid the initial load of x. Of course, if B_2 had other successors, they might not reference x and might need another value in r_2.



A second issue, both more subtle and more problematic, arises when we try to stretch the local-allocation paradigms beyond single blocks. Consider the situation that would arise when performing bottom-up local allocation on block B_1 of the example shown in the margin. If, after the use of x in B_1, the allocator needs an additional register, it must compute the distance to the next use of x. In a single block that next reference is unique, as is its distance. With multiple successor blocks, the distance will depend on the path taken at runtime, (B_1, B_2) or (B_1, B_3, B_4). Thus, it is not well defined. Even if all the subsequent uses of x, are equidistant before allocation, local spilling in one block might increase the distances on one or more paths. As the basic metric that underlies the bottom-up local method is multivalued, the algorithm's effects become harder to understand and to justify.

The effects at block boundaries can be complex. They do not fit into a local allocator because they deal with phenomena that are entirely outside a local allocator's scope. All of these problems suggest that a different approach is needed to move beyond local allocation to regional or global allocation. Indeed, successful global allocation algorithms bear little resemblance to local ones.

SECTION REVIEW

Local register allocation looks at a single basic block. That limited context simplifies the analysis and the algorithm. This section presented both a top-down and a bottom-up algorithm for local allocation. The top-down algorithm prioritizes values by the number of references to that value in the block. It assigns the highest priority values to registers. It reserves a small set of registers to handle those values that do not receive registers. The bottom-up allocator assigns values to registers as it encounters them in a forward pass over the block. When it needs an additional register, it spills the value whose next use is farthest in the future.

The top-down and bottom-up allocators presented here differ in how they treat individual values. When the top-down algorithm allocates a register for some value, it reserves that register for the entire block. When the bottom-up algorithm allocates a register for some value, it reserves that register until it encounters a more immediate need for the register. The bottom-up algorithm's ability to use a single register for multiple values allows it to produce better allocations than the top-down algorithm. The allocation paradigms in these two algorithms begin to break down when we try to apply them to larger regions.

Review Questions

1. For each of the two allocators, answer the following questions: Which step in the allocator has the worst asymptotic complexity? How might the compiler writer limit its impact on compile time?
2. The top-down allocator aggregates frequency counts by virtual register names and performs allocation by virtual register names. Sketch an algorithm that renames virtual registers in a way that improves the results of the top-down algorithm.

13.4 GLOBAL REGISTER ALLOCATION AND ASSIGNMENT

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, and data space for the spilled values. Most allocators focus on the first of these effects—minimizing the execution time of spill code.

Global register allocators cannot guarantee an optimal solution to the problem of minimizing spill code execution time. The difference between two different allocations for the same code lies in both the number of loads, stores, and copy operations that the allocator inserts and their placement in the code. The number of operations matters, both in code space and execution time. The placement of operations matters because different blocks execute different numbers of times and those execution frequencies vary from run to run.

Global allocation differs from local allocation in two fundamental ways.

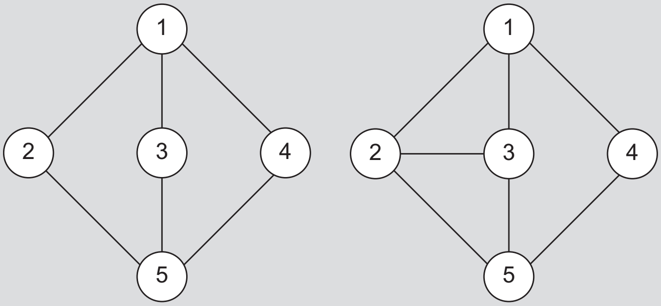
1. The structure of a global live range can be more complex than that of a local live range. A local live range is an interval in straightline code.
   A global live range is a web of definitions and uses found by taking the closure of two relationships. For a use *u* in live range LR_i, LR_i must include every definition *d* that reaches *u*. Similarly, for each definition *d* in LR_i, LR_i must include every use *u* that *d* reaches.
   Global allocators create a new name space in which each live range has a distinct name. Allocation then maps live-range names to either a physical register or a memory location.
2. Within a global live range LR_i, the distinct references may execute different numbers of times. In a local live range, all references execute once per execution of the block (unless an exception occurs). Thus, the cost of local spilling is uniform. In a global allocator, the cost of spilling depends on where the spill code occurs. The problem of choosing a value to spill is, thus, much more complex in the global case than in the local case.
   Global allocators annotate each reference with an estimated execution frequency, derived from static analysis or from profile data. Allocation then uses these annotations to guide decisions about both allocation and spilling.

Any global allocator must address both these issues. Each of these issues makes global allocation substantially more complex than local allocation.

Global allocators make decisions about both allocation and assignment. They decide, for each live range, whether or not it will reside in a register. They decide, for each enregistered live range, whether or not it can share a register with other live ranges. They choose, for each enregistered live range, a specific physical register for that live range.

GRAPH COLORING

Many global register allocators use *graph coloring* as a paradigm to model the underlying allocation 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, and the smallest such *k* for a given graph is called the graph's *chromatic number*. Consider the following graphs:



The graph on the left is two-colorable. For example, we can assign *blue* to nodes 1 and 5, and *red* to nodes 2, 3, and 4. Adding the edge (2,3), as shown on the right, makes the graph three-colorable, but not two-colorable. (Assign *blue* to nodes 1 and 5, *red* to nodes 2 and 4, and *yellow* to node 3.)

For a given graph, the problem of finding its chromatic number is NP-complete. Similarly, the problem of determining if a graph is *k*-colorable, for some fixed *k*, is NP-complete. Algorithms that use graph coloring as a paradigm to allocate resources use approximate methods to find colorings that fit the set of available resources.

To make these decisions, many compilers perform register allocation using an analogy to graph coloring. Graph-coloring allocators build a graph, called the *interference graph*, to model the conflicts between live ranges. They attempt to construct a *k*-coloring for that graph, where *k* is the number of physical registers available to the allocator. (Some physical registers, such as the ARP, may be dedicated to other purposes.) 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.

Interference graph
a graph where the nodes represent live ranges and an edge (*i,j*) indicates that LR_i and LR_j cannot share a register

Different coloring allocators handle spilling (or allocation) in different ways. We will look at top-down allocators that use high-level information to make allocation decisions and at bottom-up allocators that use low-level information to make those decisions. Before examining these two approaches, however, we explore some of the subproblems that the allocators have in common: discovering live ranges, estimating spill costs, and building an interference graph.

13.4.1 Discovering Global Live Ranges

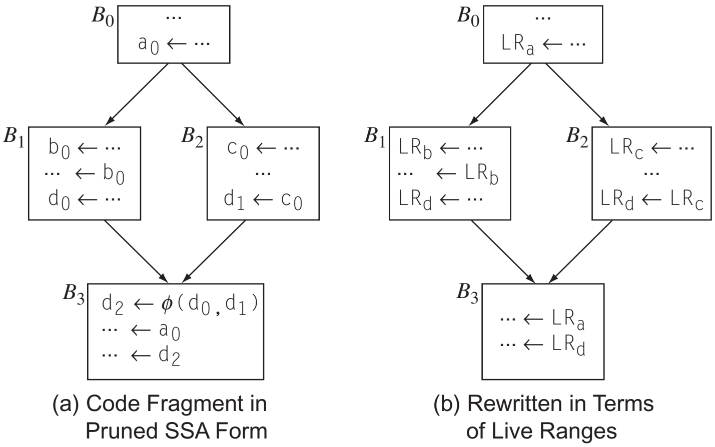
To construct live ranges, the compiler must discover the relationships that exist among different definitions and uses. The allocator must derive a name space that groups together into a single name all the definitions that reach a single use and all the uses that a single definition can reach. This suggests an approach in which the compiler assigns each definition a distinct name and merges definition names together that reach a common use. Conversion of the code into SSA form simplifies the construction of live ranges; thus, we will assume that the allocator operates on SSA form.

The SSA form of the code provides a natural starting point for this construction. Recall that in SSA form, each name is defined once, and each use refers to one definition. The ϕ-functions inserted to reconcile these two rules record the fact that distinct definitions on different paths in the control-flow graph reach a single reference. An operation that references the name defined by a ϕ-function uses the value of one of its arguments; which argument depends on how control flow reached the ϕ-function. All those definitions should reside in the same register and, thus, belong in the same live range. The ϕ-functions allow the compiler to build live ranges efficiently.

The compiler can represent global live ranges as a set of one or more SSA names.

To build live ranges from SSA form, the allocator uses the disjoint-set union-find algorithm and makes a single pass over the code. The allocator treats each SSA name, or definition, as a set in the algorithm. It examines each ϕ-function in the program, and unions together the sets associated with each ϕ-function parameter and the set for the ϕ-function result. After all the ϕ-functions have been processed, the resulting sets represent the live ranges in the code. 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.

Figure 13.3a shows a code fragment in semipruned SSA form that 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 $\{b_0\}$,



■ **FIGURE 13.3** Discovering Live Ranges.

LR_c that contains{c_0}, and LR_d that contains {d_0,d_1,d_2}. Figure 13.3b shows the code rewritten to use live-range names.

In Section 9.3.5, we saw that transformations applied to the SSA form can introduce complications into this rewriting process. If the allocator builds SSA form, uses it to find live ranges, and rewrites the code without performing other transformations, then it can simply replace SSA names with live-range names. On the other hand, if the allocator uses SSA form that has already been transformed, the rewrite process must deal with the complications described in Section 9.3.5. Since most compilers will perform allocation after instruction selection and, possibly, instruction scheduling, the code that the allocator consumes will not be in SSA form. This forces the allocator to build SSA form for the code and ensures that the rewriting process is straightforward.

13.4.2 **Estimating Global Spill Costs**

To make informed spill decisions, the global allocator needs an estimate of the cost of spilling each value. The cost of a spill has three components: the address computation, the memory operation, and an estimated execution frequency.

The compiler writer can choose where in memory to keep spilled values. Typically, they reside in a designated register-save area in the current activation record (AR) to minimize the cost of the address computation (see Figure 6.4). Storing spilled values in the AR lets the allocator generate operations such as a `loadAI` or `storeAI` relative to r_{arp} for the spill. Such operations usually avoid the need for additional registers to compute the memory address of a spilled value.

Scratchpad memory
Dedicated, noncached, local memory is sometimes called *scratchpad* memory.

Scratchpad memory is a feature in some embedded processors.

The cost of the memory operation is, in general, unavoidable. For each spilled value, the compiler must generate a store after each definition and a load before each use. As memory latencies rise, the costs of these spill operations grow. If the target processor has a fast scratchpad memory, the compiler might lower the cost of spill operations by spilling to the scratchpad memory. To make matters worse, the allocator inserts spill operations into regions where demand for registers is high. In those regions, lack of free registers may constrain the scheduler's ability to hide the memory latency. Thus, the compiler writer must hope that spill locations stay in the cache. (Paradoxically, those locations stay in the cache only if they are accessed often enough to avoid replacement—suggesting that the code is executing too many spill operations.)

Accounting for Execution Frequencies

To account for the different execution frequencies of the basic blocks in the control-flow graph, the compiler should annotate each block with an estimated execution count. The compiler can derive these estimates from profile data or from heuristics. Many compilers simply assume that each loop executes 10 times. This assumption assigns a weight of 10 to a load inside one loop, 100 to a load inside two nested loops, and so on. An unpredictable `if-then-else` would decrease the estimated frequency by half. In practice, these estimates ensure a bias toward spilling in outer loops rather than inner loops.

To estimate the cost of spilling a single reference, the allocator adds the cost of the address computation to the cost of the memory operation and multiplies that sum by the estimated execution frequency of the reference. For each live range, it sums the costs of the individual references. This requires a pass over all the blocks in the code. The allocator can precompute these costs for all live ranges, or it can wait to compute them until it discovers that it must spill at least one value.

Negative Spill Costs

A live range that contains a load, a store, and no other uses should receive a negative spill cost if the load and store refer to the same address. (Such a live range can result from transformations intended to improve the code; for example, if the use were optimized away and the store resulted from a procedure call rather than the definition of a new value.) Sometimes, spilling a live range may eliminate copy operations with a higher cost than the spill operations; such a live range also has a negative cost. Any live range with a negative spill cost should be spilled, since doing so decreases demand for registers *and* removes instructions from the code.

Infinite Spill Costs

Some live ranges are so short that spilling them does not help. Consider the short live range shown in the left margin. If the allocator tries to spill vr$_i$, it will insert a store after the definition and a load before the use, creating two new live ranges. Neither of these new live ranges uses fewer registers than the original live range, so the spill produces no benefit. The allocator should assign the original live range a spill cost of infinity, ensuring that the allocator does not try to spill it. In general, a live range should have infinite spill cost if no other live range ends between its definitions and its uses. This condition stipulates that availability of registers does not change between the definitions and uses.

```
vri       ← ···
Mem[vrj] ← vri
```

A Live Range With Infinite Spill Cost

13.4.3 Interferences and the Interference Graph

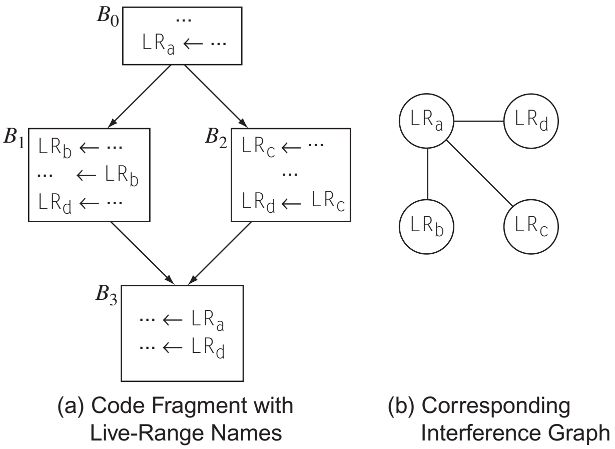
The fundamental effect that a global register allocator must model is the competition among values for space in the processor's register set. Consider two distinct live ranges,LR$_i$ and LR$_j$. If there is an operation in the program during which both LR$_i$ and LR$_j$ are live, they cannot reside in the same register. (In general, a physical register can hold just one value at a time.) We say that LR$_i$ and LR$_j$ *interfere*.

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 $I = (N,E)$, in which nodes in N represent individual live ranges and edges in E represent interferences between live ranges. Thus, an undirected edge $(n_i, n_j) \in I$ exists if and only if the corresponding live ranges LR$_i$ and LR$_j$ interfere. Figure 13.4 shows the code from Figure 13.3b along with its interference graph. As the graph shows, LR$_a$ interferes with each of the other live ranges. The rest of the live ranges, however, do not interfere with each other.

If the compiler can color I with k or fewer colors, then it can map the colors directly onto physical registers to produce a legal allocation. In the example, LR$_a$ cannot receive the same color as LR$_b$, LR$_c$, or LR$_d$ because it interferes with each of them. However, the other three live ranges can all share a single color because they do not interfere with each other. Thus, the interference graph is two-colorable, and the code can be rewritten to use just two registers.

Consider what would happen if another phase of the compiler reordered the two operations at the end of B_1. This change makes LR_b live at the definition of LR$_d$. The allocator must add the edge (LR$_b$,LR$_d$) to E, which makes



■ **FIGURE 13.4** Live Ranges and Interference.

it impossible to color the graph with just two colors. (The graph is small enough to prove this by enumeration.) To handle this graph, the allocator has two options: to use three registers, or, if the target machine has only two registers, to spill one of LR_b or LR_a before the definition of LR_d in B_1. Of course, the allocator could also reorder the two operations and eliminate the interference between LR_b and LR_d. Typically, register allocators do not reorder operations. Instead, allocators assume a fixed order of operations and leave ordering questions to the instruction scheduler (see Chapter 12.)

Building the Interference Graph

Once the allocator has built global live ranges and annotated each basic block in the code with its LiveOut set, it can construct the interference graph in a simple linear pass over each block. Figure 13.5 shows the basic algorithm. As it walks the block, from bottom to top, the allocator computes LiveNow, the set of values that are live at the current operation. (We saw LiveNow in Section 11.5.1.) At the last operation in the block, LiveOut and LiveNow must be identical. As the algorithm walks backward through the block, it adds the appropriate interference edges to the graph and updates the LiveNow set to reflect the operation's impact.

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. This definition allows the compiler to build the interference graph by adding, at each operation, an interference between the target of the operation, LR_c, and each live range that is live after the operation.

Copy operations require special treatment. 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

```
for each $LR_i$
   create a node $n_i \in N$
for each basic block b
   LiveNow ← LiveOut(b)
   for each operation $op_n$, $op_{n-1}$, $op_{n-2}$, ...$op_1$ in b
      with form $op_i$ $LR_a$ , $LR_b$ $\Rightarrow$ $LR_c$
      for each $LR_i \in$ LiveNow
         add ($LR_c$ , $LR_i$ ) to $E$
      remove $LR_c$ from LiveNow
      add $LR_a$ and $LR_b$ to LiveNow
```

■ **FIGURE 13.5** Constructing the Interference Graph.

same value and therefore can occupy the same register. Thus, the operation should not induce an edge (LR_i, LR_j) in E. If subsequent context creates an interference between these live ranges, that operation will create the edge. Likewise, a ϕ-function does not create an interference between any of its arguments and its result. Treating copies and ϕ-functions in this way creates an interference graph that precisely captures when LR_i and LR_j can occupy the same register.

To improve the allocator's efficiency, the compiler should 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, while the adjacency lists allows efficient iteration over a node's neighbors. The two-representation strategy uses more space than a single representation would, but pays off in reduced allocation time. As suggested in Section 13.2.3, the allocator can build separate graphs for disjoint register classes, which reduces the maximum graph size.

Building an Allocator

To build a global allocator based on the graph-coloring paradigm, the compiler writer needs two additional mechanisms. First, the allocator needs an efficient technique to discover k-colorings. Unfortunately, the problem of determining if a k-coloring exists for a particular graph is NP-complete. Thus, register allocators use fast approximations that are not guaranteed to find a k-coloring. Second, the allocator needs a strategy that handles the case when no color remains for a specific live range. Most coloring allocators approach this by rewriting the code to change the allocation problem. The allocator picks one or more live ranges to modify. It either spills or *splits* the chosen live ranges. Spilling turns the chosen live range into sets of tiny

Live-range splitting
If the allocator cannot keep a live range in one register, it can break the live range into smaller pieces, connected by copies or by loads and stores. The new smaller live ranges may fit into registers.

live ranges, one at each definition or use of the original live range. Splitting breaks the chosen live range into smaller, but nontrivial, pieces. In either case, the transformed code performs the same computation but has a different interference graph. If the changes are effective, the new interference graph is *k*-colorable. If they are not, the allocator must spill or split more live ranges.

13.4.4 Top-Down Coloring

A top-down graph-coloring global register allocator uses low-level information to assign colors to individual live ranges and high-level information to select the order in which it colors live ranges. To find a color for a specific live range LR$_i$ the allocator tallies the colors already assigned to LR$_i$'s neighbors in *I*. If the set of neighbors' colors is incomplete—that is, one or more colors are not used—the allocator can assign an unused color to LR$_i$. If the set of neighbors' colors is complete, then no color is available for LR$_i$ and the allocator must use its strategy for uncolored live ranges.

The top-down allocators try to color the live ranges in an order determined by some 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. These estimates are analogous to the spill costs described in Section 13.4.2. The top-down global allocator uses registers for the most important values, as identified by these rankings.

The allocator considers the live ranges in rank order and attempts to assign a color to each of them. If no color is available for a live range, the allocator invokes the spilling or splitting mechanism to handle the uncolored live range. To improve the process, the allocator can partition the live ranges into two sets—constrained live ranges and unconstrained live ranges. A live range is *constrained* if it has *k* or more neighbors—that is, it has degree $\geq k$ in *I*. Constrained live ranges are colored first, in rank order. After all constrained live ranges have been handled, the 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.

We denote "degree of LR$_i$" as LR$_i^{\circ}$. LR$_i$ is constrained if and only if LR$_i^{\circ} \geq k$.

By handling constrained live ranges first, the allocator avoids some potential spills. The alternative, working in a straight priority order, would let the allocator assign all available colors to unconstrained, but higher priority, neighbors of LR$_i$. This approach could force LR$_i$ to remain uncolored, even though colorings of its unconstrained neighbors that leave a color for LR$_i$ must exist.

Handling Spills

When the top-down allocator encounters a live range that cannot be colored, it must either spill or split some set of live ranges to change the problem. Since all previously colored live ranges were ranked higher than the uncolored live range, it makes sense to spill the uncolored live range rather than a previously colored one. The allocator can consider recoloring one of the previously colored live ranges, but it must exercise care to avoid the full generality and cost of backtracking.

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 the memory operations need registers, the allocator can reserve enough registers to handle them. (For example, a register is needed to hold the spilled value when it is loaded before a use.) The number of registers needed for this purpose is a function of the target machine's instruction set architecture. Reserving these registers simplifies spilling.

An alternative to reserving registers for spill code is to look for free colors at each definition and use; if no color is available, the allocator must retroactively spill a live range that has already colored. In this scheme, the allocator would insert the spill code, which removes the original live range and creates a new short live range, *s*. It would recompute interferences in the neighborhood of the spill site and tally the colors assigned to the neighbors of *s*. If this process does not discover an available color for *s*, the allocator spills the lowest-priority neighbor of *s*.

Of course, 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. The paradox, of course, is that reserving registers for spilling may itself cause spills by effectively lowering *k*.

Live-Range Splitting

Spilling changes the coloring problem. An uncolored live range is broken into a series of tiny live ranges, one at each definition or use. Another way to change the problem is to split an uncolored live range into new live ranges—subranges that contain several references. If the new live ranges interfere with fewer live ranges than did the original live range, they may receive colors. For example, some of the new live ranges may be unconstrained. Live-range splitting can avoid spilling the original live range at every reference; with well-chosen split points, it can isolate the portions of the live range that the allocator must spill.

The first top-down, priority-based coloring allocator, built by Chow, broke the uncolored live range into single-block live ranges, 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 the number of blocks that 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. The allocator spilled any split live ranges that remained uncolored.

13.4.5 Bottom-Up Coloring

Bottom-up graph-coloring register allocators use many of the same mechanisms as top-down global allocators. These allocators discover live ranges, build an interference graph, attempt to color it, and generate spill code when needed. The major distinction between top-down and bottom-up allocators lies in the mechanism used to order live ranges for coloring. While 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. Such an allocator constructs a linear order in which to consider the live ranges and assign colors in that order.

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

```
initialize stack to empty
while ($N \neq \emptyset$)
   if $\exists$ n $\in N$ with $n^\circ < k$
     then node $\leftarrow$ n
     else node $\leftarrow$ n picked from $N$
   remove node and its edges from $I$
   push node onto stack
```

The allocator repeatedly removes a node from the graph and places the node on a 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. Because these nodes are unconstrained, the order in which they are removed does not matter. Removing an unconstrained node decreases the degree of each of its neighbors and may make them unconstrained. The second clause, invoked only when every remaining node is constrained, picks a node using some external criteria. Any node removed by this clause has more than k neighbors and, thus, may not receive a color during the assignment phase. The loop halts when the graph is empty. At that point, the stack contains all the nodes in order of removal.

To color the graph, the allocator rebuilds the interference graph in the order represented by the stack—the reverse of the order in which the allocator removed them from the graph. It repeatedly pops a node n from the stack, inserts n and its edges back into I, and picks a color for n. The algorithm is:

```
while (stack $\neq \emptyset$)
    node $\leftarrow$ pop(stack)
    insert node and its edges into $I$
    color node
```

To pick a color for node n, the allocator tallies the colors of n's neighbors in the current approximation to I and assigns n an unused color. To pick a specific color, it can search in a consistent order each time, or it can assign colors in a round-robin fashion. (In our experience, the mechanism used for color choice has little practical impact.) If no color remains for n, it is left uncolored.

When the stack is empty, I has been rebuilt. If every node has a color, the allocator declares success and rewrites the code, replacing live-range names with physical registers. If any node remains uncolored, the allocator either spills the corresponding live range or splits it into smaller pieces. At this point, the classic bottom-up allocators rewrite the code to reflect the spills and splits and repeat the entire process—finding live ranges, building I, and coloring it. The process repeats until every node in I receives a color. Typically, the allocator halts in a couple of iterations. Of course, a bottom-up allocator could reserve registers for spilling, as the top-down allocator does. This strategy would allow it to halt after a single pass.

Why Does This Work?

The bottom-up allocator inserts each node back into the graph from which it was removed. If the reduction algorithm removes the node representing LR_i from I through its first clause (because it was unconstrained at the time of removal), then it reinserts LR_i into a graph in which it is also unconstrained. Thus, when the allocator inserts LR_i, a color must be available for LR_i. The only way that a node n can fail to receive a color is if n was removed from I using the spill metric. Such a node is inserted into a graph in which it has k or more neighbors. However, a color may still be available for n. Assume that $n^\circ > k$ when the allocator inserts it into *I*. Its neighbors cannot all have distinct colors, since they can have at most *k* colors. If they have precisely *k* colors, then the allocator finds no color for *n*. If, instead, they use fewer than *k* colors, then the allocator finds a color available for *n*.

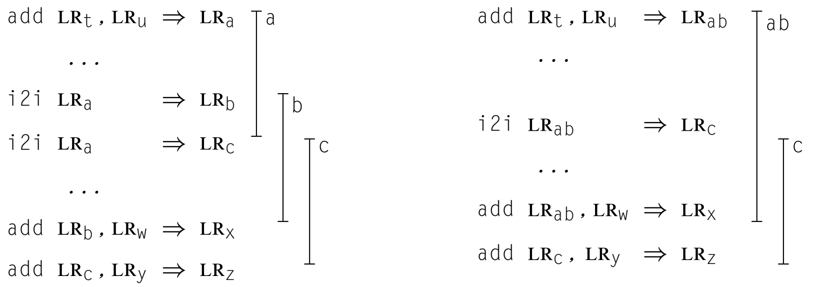
The reduction algorithm determines the order in which nodes are colored. This order is crucial, in that it determines whether or not colors are available. For nodes removed from the graph because they are unconstrained, the order is unimportant with respect to the remaining nodes. The order may be important with respect to nodes already on the stack; after all, the current node may have been constrained until some of the earlier nodes were removed. For nodes removed from the graph using the `else` clause, the order is crucial. This clause executes only when every remaining node is constrained. Thus, the remaining nodes form one or more heavily connected subgraphs of *I*.

The heuristic used by the else clause to pick a node is often called the *spill metric*. The original bottom-up graph-coloring allocator, built by Chaitin et al., used a simple spill metric. It picked a node that minimized the ratio of $\frac{cost}{degree}$, where *cost* is the estimated spill cost and *degree* is the node's degree in the current graph. This metric balances between spill cost and the number of nodes whose degree will decrease.

Other spill metrics have been tried. These include $\frac{cost}{degree^2}$, which emphasizes the impact on neighbors; straight cost, which emphasizes runtime speed; and counting the spill operations, which decreases code size. The first two, $\frac{cost}{degree}$ and $\frac{cost}{degree^2}$, attempt to balance cost and impact; the latter two, cost and spill operations, aim to optimize specific criteria. In practice, no single heuristic dominates the others. Since the actual coloring process is fast relative to building *I*, the allocator can try several colorings, each using a different spill metric, and retain the best result.

13.4.6 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. Consider the operation `i2i` $\text{LR}_i \Rightarrow \text{LR}_j$. If LR_i and LR_j do not otherwise interfere, the operation can be eliminated and all references to LR_j rewritten to use LR_i. Combining these live ranges has several beneficial effects. It eliminates the copy operation, making the code smaller and, potentially, faster. It reduces the degree of any LR_i that interfered with both LR_i and LR_j. It shrinks the set of live ranges, making *I* and many of the data structures



```
add  LRt , LRu  ⇒  LRa
      ...
i2i  LRa        ⇒  LRb
i2i  LRa        ⇒  LRc
      ...
add  LRb , LRw  ⇒  LRx
add  LRc , LRy  ⇒  LRz
```

(a) Before Coalescing

```
add  LRt , LRu   ⇒  LRab
      ...
i2i  LRab        ⇒  LRc
      ...
add  LRab , LRw  ⇒  LRx
add  LRc , LRy   ⇒  LRz
```

(b) After Coalescing LR_a and LR_b

■ **FIGURE 13.6** Coalescing Live Ranges.

related to I smaller. (In his thesis, Briggs shows examples where coalescing eliminates up to one-third of the live ranges.) Because these effects help in allocation, compilers often perform coalescing before the coloring stage in a global allocator.

Figure 13.6 shows an example. The original code appears in panel a, with lines to the right of the code that indicate the regions where each of the relevant values, LR_a, LR_b, and LR_c are live. Even though LR_a overlaps both LR_b and LR_c, it interferes with neither of them because the source and destination of a copy do not interfere. Since LR_b is live at the definition of LR_c, they do interfere. Both copy operations are candidates for coalescing.

Figure 13.6b shows the result of coalescing LR_a and LR_b to produce LR_{ab}. Since LR_c is defined by a copy from LR_{ab}, they do not interfere. Combining LR_a and LR_b to form LR_{ab} lowered the degree of LR_c. In general, coalescing two live ranges cannot increase the degrees of any of their neighbors. It can decrease their degrees or leave their degrees unchanged, but it cannot increase their degrees.

To perform coalescing, the allocator walks each block and examines each copy operation in the block. Consider a copy i2i $LR_i \Rightarrow LR_j$. If LR_i and LR_j do not interfere $(LR_i, LR_j) \notin E$, the allocator combines them, eliminates the copy, and updates I to reflect the combination. The allocator can conservatively update I by moving all edges from the node for LR_j to the node for LR_i—in effect, using LR_i as LR_{ij}. This update is not precise, but it lets the allocator continue coalescing. In practice, allocators coalesce every live range allowed by I, then rewrite the code, rebuild I, and try again. The process typically halts after a couple of rounds of coalescing.

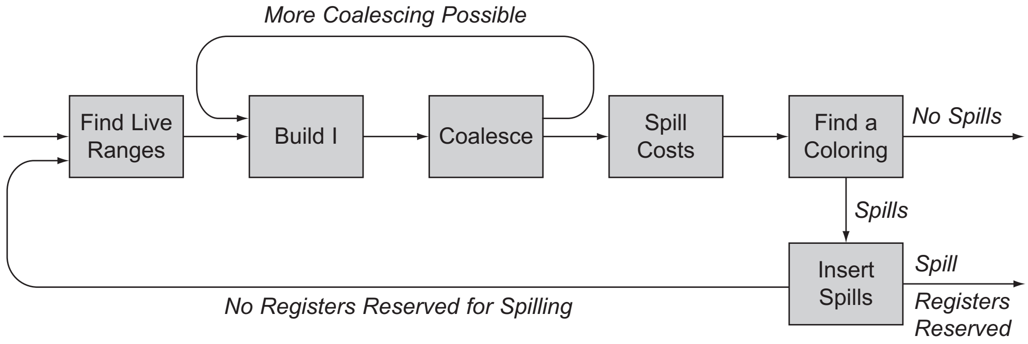
The example illustrates the imprecision inherent in this conservative update to *I*. The update would leave an interference between LR_{ab} and LR_c when, in fact, that interference does not exist. Rebuilding *I* from the transformed code produces the precise interference graph, with no edge between LR_{ab} and LR_c, and allows the allocator to coalesce LR_{ab} and LR_c.

Because coalescing two live ranges can prevent subsequent coalescing of other live ranges, the order of coalescing matters. In principle, the compiler should coalesce the most frequently executed copies first. Thus, the allocator might coalesce copies in order by the loop nesting depth of the block where the copies are found. To implement this, the allocator can consider the basic blocks in order from most deeply nested to least deeply nested.

In practice, the cost of building the interference graph for the first round of coalescing dominates the overall cost of the graph-coloring allocator. Subsequent passes through the build-coalesce loop process a smaller graph and, therefore, run more quickly. To reduce the cost of coalescing, the compiler can build a subset of the reduced interference graph—one that only includes live ranges involved in a copy operation. This observation applies the insight from semipruned SSA form to interference graph construction—only include names that matter.

13.4.7 Comparing Top-Down and Bottom-Up Global Allocators

Both the top-down and the bottom-up coloring allocators have the same basic structure, shown in Figure 13.7. They find live ranges, build the inter-



■ FIGURE 13.7 Structure of the Coloring Allocators.

ference graph, coalesce live ranges, compute spill costs on the coalesced version of the code, and attempt a coloring. The build-coalesce process is repeated until it finds no more opportunities. After coloring, one of two situations occurs. If it assigns every live range a color, then it rewrites the code using physical register names, and allocation terminates. If some live ranges remain uncolored, then it inserts spill code.

If the allocator has reserved registers for spilling, then it uses those registers in the spill code, rewrites the colored registers with their physical register names, and the process terminates. Otherwise, the allocator invents new virtual register names to use in spilling and inserts the necessary loads and stores to accomplish the spills. This changes the coloring problem slightly, so the entire allocation process is repeated on the transformed code. When each live range has a color, the allocator maps colors onto real registers and rewrites the code in its final form.

Of course, a top-down allocator could adopt the spill-and-iterate philosophy used in the bottom-up allocator. This would eliminate the need to reserve registers for spilling. Similarly, a bottom-up allocator could reserve several registers for spilling and eliminate the need for iterating the entire allocation process. Spill-and-iterate trades additional compile time for an allocation that, potentially, uses less spill code. Reserving registers produces an allocation that, potentially, contains more spills but requires less compile time to produce.

The top-down allocator uses its priority ranking to order all the constrained nodes. It colors the unconstrained nodes in arbitrary order, since the order cannot change the fact that they receive a color. The bottom-up allocator constructs an order in which most nodes are colored in a graph where they are unconstrained. Every node that the top-down allocator classifies as unconstrained is colored by the bottom-up allocator, since it is unconstrained in the original graph and in each graph derived by removing nodes and edges from *I*. The bottom-up allocator also classifies some nodes as unconstrained that the top-down allocator treats as constrained. These nodes may also be colored in the top-down allocator; there is no clear way of comparing their performance on these nodes without implementing both algorithms and running them.

The truly hard-to-color nodes are those that the bottom-up allocator removes from the graph with its spill metric. The spill metric is invoked only when every remaining node is constrained. These nodes form a strongly connected subgraph of *I*. In the top-down allocator, these nodes will be colored in an order determined by their rank or priority. In the bottom-up allocator,

LINEAR SCAN ALLOCATION

Linear scan allocators begin from the assumption that they can represent global live ranges with a simple interval [*i,j*] as we did in local allocation. This representation overestimates the extent of the live range to ensure that it includes both the earliest and the latest operation where the live range is live. The overestimate ensures that the resulting interference graph is an interval graph.

Interval graphs are much simpler than the general graphs that arise in global register allocation; for example, the interference graph of a single block is always an interval graph. From a complexity standpoint, interval graphs offer advantages to the allocator. While the problem of determining if an arbitrary graph is *k*-colorable is NP-complete, the same problem is solvable in linear time on an interval graph.

The interval representation is less expensive to build than the precise interference graph. Interval graphs lends themselves to allocation algorithms, such as the bottom-up local algorithm, that are simpler than the global allocators. Because both allocation and assignment can be performed in a single linear pass over the code, this approach is called *linear scan allocation*.

Linear scan allocators avoid building the complex precise global interference graph—the most expensive step in graph-coloring global allocators—as well as the $\mathbf{O}(N^2)$ loop to choose spill candidates. Thus, they use much less compile time than do global graph-coloring allocators. In some applications, such as just-in-time compilers (JITS), the tradeoff between speed of allocation and increase in spill code makes these linear scan allocators attractive.

Linear scan allocation has all of the subtlety seen in the global allocators. For example, using the top-down local algorithm in a linear scan allocator spills a live range everywhere it occurs, while using the bottom-up local algorithm spills it at precisely those points where the spill is needed. The imprecise notion of interference means that these allocators must use other mechanisms to coalesce copies.

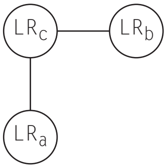
the spill metric uses that same ranking, moderated by a measurement of how many other nodes have their degree lowered by each choice. Thus, the top-down allocator chooses to spill low-priority, constrained nodes, while the bottom-up allocator spills nodes that are still constrained after all unconstrained nodes have been removed. From this latter set it picks nodes that minimize the spill metric.

13.4.8 Encoding Machine Constraints in the Interference Graph

Register allocation must deal with idiosyncratic properties of the target machine and its calling convention. Some of the constraints that arise in practice can be encoded in the coloring process.

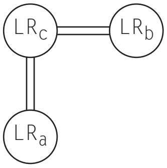
Multiregister Values

Consider a target machine that requires an aligned pair of adjacent registers for each double-precision floating-point value and a program with two single-precision live ranges LR_a and LR_b and one double-precision live range LR_c.



With interferences (LR_a, LR_c) and (LR_b, LR_c), the techniques described in Section 13.4.3 produce the graph shown in the margin. Three registers, r_0, r_1, and r_2, with a single aligned pair, (r_0, r_1)), should suffice for this graph. LR_a and LR_b can share r_2, leaving the pair (r_0, r_1) for LR_c. Unfortunately, this graph does not adequately represent the actual constraints on allocation.

Given $k = 3$, the bottom-up coloring allocator assigns colors in arbitrary order since no node has degree $\geq k$. If the allocator considers LR_c, first it will succeed, since (r_0, r_1) is free to hold LR_c. If either LR_a or LR_b is colored first, the allocator might use either r_0 or r_1, creating a situation in which the aligned register pair is not available for LR_c.



To force the desired order, the allocator can insert two edges to represent an interference with a value that needs two registers. This produces the graph at left. With this graph and $k = 3$, the bottom-up allocator must remove one of LR_a or LR_b first, since LR_c has degree 4. This ensures that two registers are available for LR_c.

The doubled edges produce a correct allocation because they match the degree of nodes that interfere with LR_c with the actual resource requirements. It does not ensure that an adjacent pair is available for LR_c. Poor assignment can leave LR_c without a pair. For example, in the coloring order LR_a, LR_c, LR_b, the allocator might assign LR_a to r_1. The compiler writer might bias the coloring order in favor of LR_c by choosing single-register values first among unconstrained nodes (the first clause in the graph-reduction algorithm). Another approach the allocator can take is to perform limited recoloring among LR_c's neighbors if an appropriate pair is not available when it tries to assign colors.

Specific Register Placement

The register allocator must also deal with requirements for the specific placement of live ranges. These constraints arise from several sources. The linkage convention dictates the placement of values that are passed in registers; this can include the ARP some or all of the actual parameters, and the return value. Some operations may require their operands in particular registers; for example, the short unsigned multiply on the Intel x86 machines always writes its result into the ax register.

As an example of the complications that arise from assigned registers in the procedure linkage, consider the typical convention on a PowerPC processor. By convention, the return value of a function is left in r_3. Suppose that the code being compiled has a function call and that the code represents the return value as vr_i. The allocator can force vr_i into r_3 by adding edges from vr_i to each physical register except r_3; this modification of the interference graph ensures that the color corresponding to r_3 is the only color available for vr_i. This solution, however, can overconstrain the interference graph.

To see the problem, assume that the code being compiled has two function calls and that the code represents those return values as vr_i and vr_j. If vr_i is live across the other call, the final code cannot keep both vr_i and vr_j in r_3. Constraining both virtual registers to map into r_3 will force one or both of them to spill.

The solution to this problem is to rely on code shape. The compiler can create a short live range for the return value at each call; say it uses vr_1 at the first call and vr_2 at the second call. It can constrain both vr_1 and vr_2 so that they map exclusively into r_3. It can add copy operations, $vr_1 \Rightarrow vr_i$ and $vr_2 \Rightarrow vr_j$. This approach creates correct code that decouples vr_i and vr_j from r_3. Of course, the allocator must constrain the coalescing mechanism to avoid combining live ranges with conflicting physical register constraints; in practice, the compiler might avoid coalescing any live range that has explicit interferences with physical registers.

$vr_1 \leftarrow vr_2 \times vr_3$

$vr_5 \leftarrow vr_1 \times vr_4$

mov ax, vr_2
imul vr_3
imul vr_4

As an example of the physical register constraints that an ISA can impose, consider the one-address integer multiply operation on an Intel x86 processor. It use the ax register as its implicit second argument and as its result register. Consider mapping the IR sequence shown in the margin into x86 code. The compiler might constrain vr_2, vr_1, and vr_5 so that they map into the ax register. In that case, the process might produce a code sequence similar to the pseudo-assembly code at the left, with the virtual register names, vr_i, replaced with their actual runtime locations. As long as the live ranges mapped to ax are short, this strategy can produce high-quality code. Again, coalescing must be constrained on any live ranges that overlap other operations that require ax.

SECTION REVIEW

Global register allocators consider the longer and more complex live ranges that arise from control-flow graphs that contain multiple blocks. Accordingly, global allocation is harder than local allocation. Most global allocators operate by analogy to graph coloring. The allocator builds a graph that represents the interferences between live ranges, and then it attempts to find a *k*-coloring for that graph, where *k* is the number of registers available to the allocator.

Graph-coloring allocators vary in the precision in their definition of a live range, in the precision with which they measure interference, in the algorithm used to find a *k*-coloring, and in the technique that they use to select values for spilling or splitting. In general, these allocators produce reasonable allocations with acceptable amounts of spill code. The major opportunities for improvement appear to be in the areas of spill choice, spill placement, and in live-range splitting.

Review Questions

1. The original top-down, priority-driven register allocator used a different notion of interference than that presented in Section 13.4.3. It added an edge $(\mathrm{LR}_i, \mathrm{LR}_j)$ to the graph if LR_i and LR_j were live in the same basic block. What impact would that definition have on the allocator? On register coalescing?
2. The bottom-up global allocator chooses values to spill by finding the value that minimizes some ratio, such as $\frac{\textit{spill cost}}{\textit{degree}}$. When the algorithm runs, it sometimes must choose several live ranges to spill before it makes any other live range unconstrained. Explain how this situation can happen. Can you envision a spill metric that avoids this problem?

13.5 ADVANCED TOPICS

Because the cost of a misstep during register allocation can be high, algorithms for register allocation have received a great deal of attention. Many variations on the basic graph-coloring allocation techniques have been published. Section 13.5.1 describes several of these approaches. Section 13.5.2 sketches another promising approach: using SSA names as live ranges in a global allocators.

13.5.1 Variations on Graph-Coloring Allocation

Many variations on these two basic styles of graph-coloring register allocation have appeared in the literature. This section describes several of these improvements. Some address the cost of allocation. Others address the quality of allocation.

Imprecise Interference Graphs

Chow's top-down, priority-based allocator used an imprecise notion of interference: live ranges LR_i and LR_j interfere if both are live in the same basic block. This makes building the interference graph faster. However, the imprecise nature of the graph overestimates the degree of some nodes and prevents the allocator from using the interference graph as a basis for coalescing. (In an imprecise graph, two live ranges connected by a useful copy interfere because they are live in the same block.) The allocator also included a prepass to perform local allocation on values that are live in only one block.

Breaking the Graph into Smaller Pieces

If the interference graph can be separated into components that are not connected, those disjoint components can be colored independently. Since the size of the bit matrix is $\mathbf{O}(N^2)$, breaking it into independent components saves both space and time. One way to split the graph is to consider nonoverlapping register classes separately, as with floating-point registers and integer registers. A more complex alternative for large procedures is to discover clique separators, connected subgraphs whose removal divides the interference graph into several disjoint pieces. For large enough graphs, using a hash table instead of the bit matrix may improve both speed and space.

Conservative Coalescing

Conservative coalescing
a form of coalescing that only combines LR_i and LR_j if LR_{ij} receives a color

When the allocator coalesces two live ranges, LR_i and LR_j, the new live range, LR_{ij}, may be more constrained than either LR_i or LR_j. If LR_i and LR_j have distinct neighbors, then $\text{LR}^{\circ}_{ij} > \max(\text{LR}^{\circ}_{i},\text{LR}^{\circ}_{j})$. If $\text{LR}^{\circ}_{ij} < k$, then creating LR_{ij} is strictly beneficial. However, if $\text{LR}^{\circ}_{i} < k$ and $\text{LR}^{\circ}_{j} < k$, but $\text{LR}^{\circ}_{ij} \geq k$, then coalescing LR_i and LR_j can make I harder to color without spilling. To avoid this problem, the compiler writer can use a limited form of coalescing called *conservative coalescing*. In this scheme, the allocator only combines LR_i and LR_j if LR_{ij} has fewer than k neighbors of "significant" degree—that is, neighbors in I that themselves have k or more neighbors. This restriction ensures that coalescing LR_i and LR_j does not make I harder to color.

If the allocator uses conservative coalescing, another improvement is possible. When the allocator reaches a point at which every remaining live range is constrained, the basic algorithm selects a spill candidate. An alternative approach is to reapply coalescing at this point. Live ranges that were not coalesced because of the degree of the resulting live range may well coalesce in the reduced graph. Coalescing at this point may reduce the degree of nodes that interfere with both the source and destination of the copy. This style of *iterated coalescing* can remove additional copies and reduce the degrees of nodes. It may create one or more unconstrained nodes and allow coloring to proceed. If iterated coalescing does not create any unconstrained nodes, spilling proceeds as before.

Biased coloring is another approach to coalescing copies without making the graph harder to color. In this approach, the allocator tries to assign the same color to live ranges that are connected by a copy. In picking a color for LR_i, it first tries colors that have been assigned to live ranges connected to LR_j by a copy operation. If it can assign them both the same color, the allocator eliminates the copy. With a careful implementation, this adds little or no expense to the color selection process.

Spilling Partial Live Ranges

As described, both approaches to global allocation spill entire live ranges. This approach can lead to overspilling if the demand for registers is low through most of the live range and high in a small region. More sophisticated spilling techniques find the regions where spilling a live range is productive—that is, the spill frees a register in a region where a register is truly needed. The splitting scheme described for the top-down allocator achieved this result by considering each block in the spilled live range separately. A bottom-up allocator can achieve similar results by spilling only in the region where interference occurs. One technique, called *interference-region spilling*, identifies a set of live ranges that interfere in the region of high demand and limits spilling to that region. The allocator can estimate the cost of several spilling strategies for the interference region and compare those costs against the standard spill-everywhere approach. By letting the alternatives compete on an estimated-cost basis, the allocator can improve overall allocation.

Live-Range Splitting

Breaking a live range into pieces can improve the results of coloring-based register allocation. In principle, splitting harnesses two distinct effects. If the split live ranges have lower degrees than the original one, they may be easier to color—possibly even unconstrained. If some of the split live ranges have high degree and, therefore, spill, then splitting may prevent spilling other portions of the same live range that have lower degree. As a final, pragmatic effect, splitting introduces spills at the points where the live range is broken. Careful selection of the split points can control the placement of some spill code—for example, outside loops rather than inside them.

Many approaches to splitting have been tried. Section 13.4.4 describes one that breaks a live range into blocks and coalesces them back together if doing so does not change the allocator's ability to assign a color. Several approaches that use properties of the control-flow graph to choose splitting points have been tried. Briggs showed that many have been inconsistent [45]; however, two particular techniques show promise. A method called *zero-cost splitting* capitalizes on `nops` in the instruction schedule to split live ranges and improve both allocation and scheduling. A technique called *passive splitting* uses a directed interference graph to determine where splits should occur and selects between splitting and spilling based on their estimated costs.

Rematerialization

Some values cost less to recompute than to spill. For example, small integer constants should be recreated with a load immediate rather than being retrieved from memory with a load. The allocator can recognize such values and rematerialize them rather than spill them.

Modifying a bottom-up graph-coloring allocator to perform rematerialization takes several small changes. The allocator must identify and tag SSA names that can be rematerialized. For example, any operation whose arguments are always available is a candidate. It can propagate these rematerialization tags over the code using the constant-propagation algorithm described in Chapter 9. In forming live ranges, the allocator should only combine SSA names that have identical rematerialization tags.

The compiler writer must make the spill-cost estimation handle rematerialization tags correctly, so that these values have accurate spill-cost estimates. The spill-code insertion process must also examine the tags and generate the appropriate lightweight spills for rematerializable values. Finally, the allocator should use conservative coalescing to avoid prematurely combining live ranges with distinct rematerialization tags.

Ambiguous Values

In code that makes heavy use of ambiguous values, whether derived from source-language pointers, array references, or object references whose class cannot be determined at compile time, the allocator's ability or inability to keep such values in registers is a serious performance issue. To improve allocation of ambiguous values, several systems have included transformations that rewrite the code to keep unambiguous values in scalar local variables, even when their "natural" home is inside an array element or a pointer-based structure. Scalar replacement uses array-subscript analysis to identify reuse of array-element values and to introduce scalar temporary variables that hold reused values. Register promotion uses data-flow analysis of pointer values to determine when a pointer-based value can safely be kept in a register throughout a loop nest and to rewrite the code so that the value is kept in a newly introduced temporary variable. Both of these transformations encode the results of analysis into the shape of the code, making it obvious to the register allocator that these values can be kept in registers. These transformations can increase the demand for registers. In fact, promoting too many values can produce spill code whose cost exceeds the the cost of the memory operations that the transformation is intended to avoid. Ideally, these techniques should be integrated into the allocator in which realistic estimates of the demand for registers can be used to determine how many values to promote.

13.5.2 **Global Register Allocation over SSA Form**

The complexity of global register allocation shows up in many ways. In the graph-coloring formulation, that complexity exhibits itself in the fact that the problem of determining if a k-coloring of a general graph exists is NP-complete. For restricted classes of graphs, the coloring problem has polynomial-time solutions. For example, the interval graphs generated by a basic block can be colored in time linear in the size of the graph. To capitalize on this fact, linear scan allocators approximate global live ranges with simple intervals that produce an interval graph.

If the compiler builds an interference graph from SSA names rather than live ranges, the resulting graph is a *chordal graph*. The problem of k-coloring a chordal graph can be solved in $\mathbf{O}(|V| + |E|)$ time. This observation has sparked interest in global register allocation over the SSA form of the code.

Chordal graph
a graph in which every cycle of more than three nodes has a *chord*—an edge that joins two nodes that are not adjacent in the cycle

Working from SSA form simplifies some parts of the register allocator. The allocator can compute an optimal coloring for its interference graph, rather than relying on heuristic approaches to coloring. The optimal coloring may use fewer registers than the heuristic coloring would.

If the graph needs more than k colors, the allocator still must spill one or more values. While SSA form does not lower the complexity of spill choice, it may offer some benefits. Global live ranges tend to have longer lifetimes than SSA names, which are broken by ϕ-functions at appropriate places in the code, such as loop headers and blocks that follow loops. These breaks give the allocator the chance to spill values over smaller regions than it might have with global live ranges.

Unfortunately, SSA-based allocation leaves the code in SSA form. The allocator, or a postpass, must translate out of SSA form, with all of the complications discussed in Section 9.3.5. That translation may increase demand for registers. (If the translation must break a cycle of concurrent copies, it needs an additional register to do so.) An SSA-based allocator must be prepared to handle this situation.

Equally important, that translation inserts copy operations into the code; some of those copies may be extraneous. The allocator cannot coalesce away copies that implement the flow of values corresponding to a ϕ-function; to do so would destroy the chordal property of the graph. Thus, an SSA-based allocator would probably use a coalescing algorithm that is not based on the interference graph. Several strong algorithms exist.

It is difficult to asses the merits of an SSA-based allocator versus an allocator based on traditional global live ranges. The SSA-based allocator has the potential to obtain a better coloring than the traditional allocator, but it does so on a different graph. Both allocators must address the problems of spill choice and spill placement, which may contribute more to performance than the actual coloring. The two allocators use different techniques for copy coalescing. As with any register allocator, the actual low-level details of the implementation will matter.

13.6 SUMMARY AND PERSPECTIVE

Because register allocation is an important part of a modern compiler, it has received much attention in the literature. Strong techniques exist for both local and global allocation. Because many of the underlying problems are NP-hard, the solutions tend to be sensitive to small decisions, such as how ties between identically ranked choices are broken.

Progress in register allocation has come from the use of paradigms that provide intellectual leverage on the problem. Thus, graph-coloring allocators have been popular, not because register allocation is identical to graph coloring, but rather because coloring captures some of the critical aspects of the global allocation problem. In fact, many of the improvements to coloring allocators have come from attacking the points where the coloring paradigm does not accurately reflect the underlying problem, such as better cost models and improved methods for live-range splitting. In effect, these improvements have made the paradigm more closely fit the real problem.

CHAPTER NOTES

Register allocation dates to the earliest compilers. Backus reports that Best invented the bottom-up local algorithm in the mid-1950s, during the development of the original FORTRAN compiler [26, 27]. Best's algorithm has been rediscovered and reused in many contexts over the years [36, 117, 181, 246]. Its best-known incarnation is as Belady's offline page-replacement algorithm [36]. The complications that arise from having a combination of clean values and dirty values are described by Horwitz [196] and by Kennedy [214]. Liberatore et al. suggest spilling clean values before dirty values as a practical compromise [246]. The example on page 688 and 689 was suggested by Ken Kennedy.

The connection between graph coloring and storage-allocation problems was suggested by Lavrov [242] many years earlier; the Alpha project used coloring to pack data into memory [140, 141]. The first complete graph-coloring allocator to appear in the literature was an allocator built by Chaitin and his colleagues for IBM's PL.8 compiler [73, 74, 75]. Schwartz describes early algorithms by Ershov and by Cocke [310] that focus on reducing the number of colors and ignore spilling.

Top-down graph coloring begins with Chow [81, 82, 83]. His implementation worked from a memory-to-memory model, used an imprecise interference graph, and performed live-range splitting as described in Section 13.4.4. It uses a separate optimization pass to coalesce copies [81]. Chow's algorithm was used in several prominent compilers. Larus built a top-down, priority-based allocator for SPUR LISP that used a precise interference graph and operated from a register-to-register model [241]. The top-down allocation in Section 13.4.4 roughly follows Larus' plan.

The bottom-up allocator in Section 13.4.5 follows Chaitin's plan with Briggs' modifications [51, 52, 56]. Chaitin's contributions include the fundamental definition of interference and the algorithms for building the interference graph, for coalescing, and for handling spills. Briggs presented an SSA-based algorithm for live range construction, an improved coloring heuristic, and several approaches to live-range splitting [51]. Other significant improvements in bottom-up coloring have included better methods for spilling [37, 38], rematerialization of simple values [55], stronger coalescing methods [158, 280], and methods for live-range splitting [98, 106, 235]. Gupta, Soffa, and Steele suggested shrinking the graph with clique separators [175], while Harvey proposed splitting it by register classes [101].

Chaitin, Nickerson, and Briggs all discuss adding edges to the interference graph to model specific constraints on assignment [54, 75, 275]. Smith et al. present a clear treatment of how to handle register classes [319]. Both scalar replacement [67, 70] and register promotion [250, 253, 306] rewrite the code to increase the set of values that the allocator can keep in registers.

The observation that SSA names form a chordal graph was made independently by several authors [58, 177, 283]. Both Hack and Bouchez built on the original observation with in-depth treatments of SSA-based global allocation [47, 176].

EXERCISES

Section 13.3

1. Consider the following ILOC basic block. Assume that r_{arp} and r_i are live on entry to 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
jmp                 →  L_003
```

   a. Show the result of using the top-down local algorithm on it to allocate registers. Assume a target machine with four registers.

   b. Show the result of using the bottom-up local algorithm on it to allocate registers. Assume a target machine with four registers.

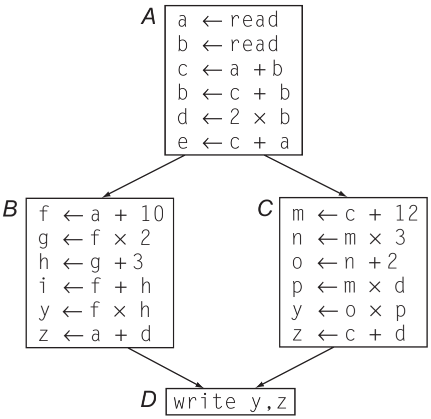
2. The top-down local allocator is somewhat naive in its handling of values. It allocates one value to a register for the entire basic block.

   a. An improved version might calculate live ranges within the block and allocate values to registers for their live ranges. What modifications would be necessary to accomplish this?

   b. A further improvement might be to split the live range when it cannot be accommodated in a single register. Sketch the data structures and algorithmic modifications that would be needed to (1) break a live range around an instruction (or range of instructions) where a register is not available and to (2) reprioritize the remaining pieces of the live range.

   c. With these improvements, the frequency count technique should generate better allocations. How do you expect your results to compare with using the bottom-up local algorithm? Justify your answer.

3. Consider the following control-flow graph:



```
A
a ← read
b ← read
c ← a +b
b ← c + b
d ← 2 × b
e ← c + a

B
f ← a + 10
g ← f × 2
h ← g +3
i ← f + h
y ← f × h
z ← a + d

C
m ← c + 12
n ← m × 3
o ← n +2
p ← m × d
y ← o × p
z ← c + d

D
write y,z
```

Assume that `read` returns a value from external media and that `write` transmits a value to external media.

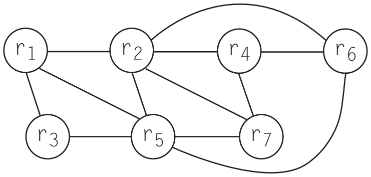
a. Compute the LiveIn and LiveOut sets for each block.

b. Apply the bottom-up local algorithm to each block, *A*, *B*, and *C*. Assume that three registers are available to the computation. If block *b* defines a name *n* and *n* ∈ LiveOut(*b*), the allocator must store *n* back to memory so that its value is available in subsequent blocks. Similarly, if block *b* uses name *n* before any local definition of *n*, it must load *n*'s value from memory. Show the resulting code, including all loads and stores.

c. Suggest a scheme that would allow some of the values in LiveOut(*A*) to remain in registers, avoiding their initial loads in the successor blocks.

4. Consider the following interference graph:

Section 13.4



Assume that the target machine has just three registers.

a. Apply the bottom-up global coloring algorithm to the graph. Which virtual registers are spilled? Which are colored?

b. Does the choice of spill node make a difference?

c. Earlier coloring allocators spilled any live range that is constrained when it is selected. Rather than applying the algorithm shown in Figure 13.8, they used the following method:

```
initialize stack to empty
while (N ≠ ∅)
   if ∃ n∈N with n° < k then
      remove n and its edges from I
      push n onto stack
   else
      pick a node n from N
      mark n to be spilled
```

If this marks any node for spilling, the allocator inserts spill code and repeats the allocation process on the modified program. If no node is marked for spilling, it proceeds to assign colors in the manner described for the bottom-up global allocator.

What happens when you apply this algorithm to the example interference graph? Does the mechanism used to choose a node for spilling change the result?

5. After register allocation, a careful analysis of the code may discover that, in some stretches of the code, there are unused registers. In a bottom-up, graph-coloring, global allocator, this occurs because of detailed shortcomings in the way that live ranges are spilled.

a. Explain how this situation can arise.

b. How might the compiler discover if this situation occurs and where it occurs?

c. What might be done to use these unused registers, both within the global framework and outside of it?

6. When a graph-coloring allocator reaches the point where no color is available for a particular live range, LR_i, it spills or splits that live range. As an alternative, it might attempt to recolor one or more of LR_i's neighbors. Consider the case where $(LR_i, LR_j) \in I$ and $(LR_i, LR_k) \in I$, but $(LR_j, LR_k) \notin I$. If LR_j and LR_k have already been colored, and have received different colors, the allocator might be able to recolor one of them to the other's color, freeing up a color for LR_i.

a. Sketch an algorithm that discovers if a legal and productive recoloring exists for LR_i.

b. What is the impact of your technique on the asymptotic complexity of the register allocator?

c. If the allocator cannot recolor LR_k to the same color as LR_j because one of LR_k's neighbors has the same color as LR_j, should the allocator consider recursively recoloring LR_k's neighbors? Explain your rationale.

7. The description of the bottom-up global allocator suggests inserting spill code for *every* definition and use in the spilled live range. The top-down global allocator first breaks the live range into block-sized pieces, then combines those pieces when the result is unconstrained and, finally, assigns them a color.

a. If a given block has one or more free registers, spilling a live range multiple times in that block is wasteful. Suggest an improvement to the spill mechanism in the bottom-up global allocator that avoids this problem.

b. If a given block has too many overlapping live ranges, then splitting a spilled live range does little to address the problem in that block. Suggest a mechanism (other than local allocation) to improve the behavior of the top-down global allocator inside blocks with high demand for registers.

8. Consider spilling in the bottom-up global allocator. When the allocator must spill, it chooses the value that minimizes the ratio $\frac{\textit{spill cost}}{\textit{degree}}$.

In a procedure with a single long block, or a single long block inside a loop nest, the spill cost for a live range approximates its frequency count. Thus, a live range that is heavily used at the beginning and ends of the long block, but unreferenced in the middle, ties up a register for the entire block.

How might you modify the bottom-up allocator so that its spill behavior on long blocks more closely resembled the behavior of the bottom-up local algorithm than the top-down local algorithm?