a condition-code register has similar effects. Thus, general-purpose registers should be allocated after the other register classes.

### 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. In optimization, methods that handle a single basic block are termed *local* methods, so these algorithms are local register-allocation techniques. The allocator takes as input a single basic block that incorporates a register-to-register memory model.

To simplify the discussion, we assume that the program starts and ends with the block; it inherits no values from blocks that executed earlier and leaves behind no values for blocks that execute later. Our input program will use only a single class of general-purpose registers. Our target machine will support a single set of \( k \) general-purpose registers.

The code shape encodes information about which values can legally reside in a register for nontrivial amounts of time. Any value that can legally reside in a register is kept in a register. The code uses as many register names as needed to encode this information, so it may name more registers than the target machine has. For this reason, we call these preallocation registers *virtual registers*. For a given block, the number of virtual registers that it uses is \( \text{MaxVR} \).

The basic block consists of a series of \( N \) three-address operations \( o_1, o_2, o_3, \ldots, o_n \). Each operation, \( o_i \), has the form \( op_1 \, vr_{r_1}, vr_{r_2} \Rightarrow vr_{r_3} \). The notation \( vr \) denotes the fact that these are virtual registers, rather than physical registers. 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 \( \text{MaxVR} > k \), the allocator may need to insert loads and stores to fit the code into the set of \( k \) 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 will explore two approaches to this problem. The first approach counts the number of references to a value in the block and uses these frequency counts to determine which values will reside in registers. Because it relies on externally derived information—the frequency counts—to make its decisions, 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 counts the number of occurrences of each virtual register in the block and uses these frequency counts as priorities to allocate virtual registers to physical registers.

If there are more virtual registers than physical registers, the allocator must reserve several physical registers for use in computations that involve values allocated to memory. The allocator must have enough registers to address and load two operands, to perform the operation, and to store the result. The precise number of registers needed depends on the target architecture; on a typical RISC machine, the number might be two to four registers. We will refer to this machine-specific number as feasible.

To perform top-down local allocation, the compiler can apply the following simple algorithm:

1. **Compute a priority for each virtual register.** In a linear pass over the operations in the block, the allocator can tally the number of times each virtual register appears. A virtual register's count becomes its priority.

2. **Sort the virtual registers into priority order.** If blocks are reasonably small, it can use a bucket sort, since the scores must fall within a small range, between zero and a small multiple of the block length.

3. **Assign registers in priority order.** The first $k$ — feasible virtual registers are assigned physical registers.

4. **Rewrite the code.** In a second walk over the code, the allocator can rewrite the code. References to virtual registers with assigned physical registers are rewritten with the physical register names. Any reference to a virtual register with no physical register is replaced with a reference to a reserved temporary register; a load or store operation is inserted, as appropriate.

The strength of this approach is that it keeps heavily used virtual registers in physical registers. Its primary weakness lies in the approach to allocation—it dedicates a physical register to a virtual register for the entire basic block. Thus, a value that is heavily used in the first half of the block and unused in the second half of the block occupies its physical register through the second half, even though it is no longer of use. 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 transitions that occur as each operation executes. It 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 structure 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 vr1 and vr2 in order, the allocator avoids using two physical registers

```c
/* code for the allocator */
for each operation, i, in order from 1 to N where i has the form
   op vr1 vr2 ⇒ vr3
   r_x ← ensure(vr1, class(vr1))
   r_y ← ensure(vr2, class(vr2))
   if vr1 is not needed after i
     then free(r_x, class(r_y))
   if vr2 is not needed after i
     then free(r_y, class(r_y))
   r_z ← allocate(vr3, class(vr3))
   rewrite i as op r_x r_y ⇒ r_z
   if vr1 is needed after i
     then class.Next[r_z] ← dist(vr1)
   if vr2 is needed after i
     then class.Next[r_z] ← dist(vr2)
   class.Next[i] ← dist(vr3)
free(i, class)
   if (class.Free[i] ≠ true) then
     push(i, class)
     class.Name[i] ← -1
     class.Next[i] ← ∞
     class.Free[i] ← true
```

*Figure 13.1 The Bottom-Up, Local Register Allocator*
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 the result when the operation actually frees a register. Most of the complications are hidden in the routines \textit{Ensure}, \textit{Allocate}, and \textit{Free}.

The routine \textit{Ensure} is conceptually simple. It takes two arguments, a virtual register, \( vr \), holding the desired value, and a representation for the appropriate register class, \( class \). If \( vr \) already occupies a physical register, \textit{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.

\textit{Allocate} and \textit{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 on the left side of Figure 13.2. A class has \texttt{Size} physical registers, each of which is represented by a virtual register name (\texttt{Name}); an integer that indicates the distance to its next use (\texttt{Next}); and a flag indicating whether or not that physical register is currently in use (\texttt{Free}). The code on the right side of the figure initializes the class structure, using \(-1\) as an out-of-range name and \(\infty\) as the maximum possible distance. To make \textit{Allocate} and \textit{Free} efficient, the class also needs a list of free registers—the Stack in \texttt{Class}. Routines \texttt{push} and \texttt{pop} manipulate the Stack.

With this level of detail, the code for both \textit{Allocate} and \textit{Free} is straightforward. Each class maintains a stack of free physical registers. \textit{Allocate} returns a physical register from the free list of \texttt{class}, if one exists. Otherwise, it selects the value stored in \texttt{class} that is used farthest in the future, stores it, and reallocates the physical register for \( vr \). \textit{Allocate} sets the \texttt{Next} field to \(-1\), ensuring that this register will not be chosen for the other operand in the current operation. The main loop of the allocator will reset the \texttt{Next} field to its appropriate value after it finishes with the current operation. \textit{Free} pushes the register onto the stack and resets its fields in the \texttt{class} structure.

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

```c
initialize(class, size)
    class.Size ← size
>class.StackTop ← −1
    for i ← 0 to size − 1
        class.Name[i] ← −1
        class.Next[i] ← ∞
        class.Free[i] ← true
    push(i, class)
```

\textbf{Figure 13.2} Representing a Register Class in C
The function \( \text{dist}(w) \) returns the index in the block of the next reference to \( w \). The compiler can annotate each reference in the block with the appropriate value for \( \text{dist} \) by making a single backward pass over the block.

The net effect of this bottom-up technique is straightforward. Initially, it assumes that the physical registers are unoccupied and places them on a free list. For the first few operations, it satisfies demand from the free list. When the allocator needs another register and discovers that the free list is empty, it must spill a value from a register to memory. It picks the value whose next use is farthest in the future. As long as the cost of spilling a value is the same for all registers, this choice frees up the register 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 in practice. 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 choose 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.

\[
\begin{align*}
\text{store } x_2 \\
\text{load } x_3 \\
\text{load } x_2 \\
\text{Spill Dirty Value}
\end{align*}
\]

\[
\begin{align*}
\text{load } x_3 \quad \text{(overwriting } x_1 \text{)} \\
\text{load } x_1 \\
\text{Spill Clean Value}
\end{align*}
\]

If, instead, the allocator spills \( x_1 \), it produces the sequence of memory operations shown on the right—one fewer memory operation. 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 one fewer memory operation. Taking into account the distinction between clean values and dirty values makes the local allocation problem NP-hard. Still, in practice, versions of the bottom-up local allocator produce good local allocations; they tend to be better than those produced by the top-down allocator previously described.

The bottom-up local allocator differs from the top-down local one in the way that it handles individual values. The top-down allocator devotes a physical register to a virtual register for an entire block. The bottom-up allocator assigns a physical register to a virtual register for the distance between two consecutive references to the virtual register. It reconsiders that decision at each invocation of Allocate—that is, each time that it needs another register. Thus, the bottom-up algorithm can, and does, produce allocations in which a single virtual register is kept in different physical registers at different points in its lifetime. Similar behavior can be retrofitted into the top-down allocator.

13.4 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 the bottom-up local allocator. We could use lessons learned in the bottom-up local algorithm to improve the frequency-count allocator. However, local allocation does not capture the reuse of values across multiple blocks. This happens regularly in practice. Thus, the next step is to extend the scope of allocation beyond single basic 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