Engineering a Compiler

Draft of Second Edition

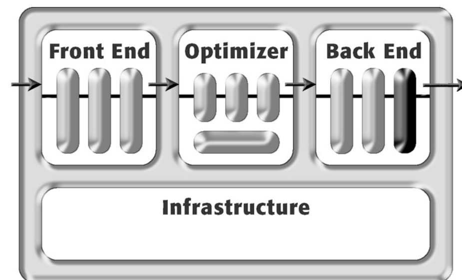
Keith D. Cooper
Linda Torczon

Rice University
Houston, Texas

Limited Copies Distributed
Reproduction requires explicit permission

Copyright 2009, Morgan-Kaufmann Publishers and the authors
All rights reserved

CHAPTER 13



Register Allocation

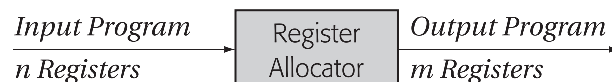
13.1 INTRODUCTION

Not in new format

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 compiled code, 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. Alternately, the value might be kept in memory between uses because the allocator cannot prove that it can safely reside in a register.

Conceptually, the register allocator takes as its input a program that uses some arbitrary number of registers.



It produces as its output an equivalent program that fits into the register set of the target machine. When the allocator cannot keep some value in a register, it must store the value to memory and load it again when it is next needed. This process is called *spilling* the value to memory.

Typically, the register allocator's goal is to make effective use of the register set provided by the target machine. This includes minimizing the number of load and store operations that execute to perform spilling. However, other goals are possible. For example, in a memory-constrained environment, the user might want the allocator to minimize the memory impact of allocation—both data memory that holds spilled values and code memory that holds spill operations.

A bad decision in the register allocator causes some value to be spilled that might otherwise reside in a register. The cost of this extra spill code rises with memory latency. Thus, the growing disparity between memory speed and processor speed in the 1990s has increased the impact of register allocation on the performance of compiled 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 scopes and global scopes.

13.2 BACKGROUND ISSUES

The register allocator takes code that is almost completely compiled as input—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 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 (see Section 5.4.3) defines many details of the problem that the allocator must address. In a registerto-register model, earlier phases in the compiler directly encode their knowledge about ambiguity of memory references into the shape of the IR program—by making all unambiguous values reside in virtual registers. Values that are memory based in the IR program 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. The IR program keeps all values in memory, and it moves them in and out of registers as they are used and defined. The allocator must determine which values can safely be kept in registers because they are unambiguous. It must then 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 needs to 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, the allocator in a compiler that uses a memory-to-memory model is an optimization that improves the performance of a legal program. The allocator decides to keep some memory-based values in registers, which makes some of the loads and stores in the program unnecessary. 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 registerto-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 set of names to the specific set of registers provided by 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, in almost any realistic formulation, NP-complete. For a single basic block, with one size of data value, optimal allocation can be done in polynomial time, as long as the cost of storing values back to memory 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 realistic memory model, or the fact that some values need not be stored back to memory, 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 do.

Register assignment, in many cases, can be solved in polynomial time. 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.

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 sixteen general-purpose registers and four floating-point registers. Modern processors may add more classes. For example, the IBM/Motorola 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 be able to allocate registers for them independently. This breaks the problem into smaller, independent components, reduces the size of the data structures, and may produce faster compile times. When two register classes overlap, however, then both classes must be modelled in a single allocation problem. The common architectural practice of keeping double-precision floating-point numbers in pairs of single-precision registers is a good example of this issue. The classes of double-precision values and single-precision values both map to the same underlying set of hardware registers. The compiler cannot allocate one of these classes without considering the other, so it must solve the joint allocation problem.

Even if the different register classes are physically and logically separate, they interact through operations that refer to registers in multiple classes. For example, for many architectures, the decision to spill a floating-point register requires the insertion of an address calculation and some memory operations; these actions use general-purpose registers and change the allocation problem for that class of registers. Thus, the compiler can make independent allocation decisions for the different classes, but those decisions can have consequences that affect allocation in other register classes. Spilling a predicate register or 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 *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 $op_i\ vr_{i_1}, vr_{i_2} \Rightarrow vr_{i_3}$. 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 $MaxVR > k$, 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 num-

ber 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 vr_{i_1} and vr_{i_2} in order, the allocator avoids using two physical registers for an

```
/* code for the allocator */
for each operation, i, in order from 1
   to N where i has the form
        op vr_i1 vr_i2 ⇒ vr_i3
   r_x ← ensure(vr_i1, class(vr_i1))
   r_y ← ensure(vr_i2, class(vr_i2))
   if vr_i1 is not needed after i
      then free(r_x,class(r_x))
   if vr_i2 is not needed after i
      then free(r_y,class(r_y))
   r_z ←allocate(vr_i3, class(vr_i3))
   rewrite i as op_i r_x,r_y ⇒ r_z
   if vr_i1 is needed after i
      then class.Next[r_x] ← dist(vr_i1)
   if vr_i2 is needed after i
      then class.Next[r_y] ← dist(vr_i2)
   class.Next[r_z] ← dist(vr_i3)

free(i,class)
   if (class.Free[i] ≠ true) then
      push(i,class)
      class.Name[i] ← −1
      class.Next[i] ← ∞
      class.Free[i] ← true

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

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 *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 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 on the left side of Figure 13.2. 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`). The code on the right side of the figure initializes the class structure, using -1 as an out-of-range name and ∞ as the maximum possible distance. To make *Allocate* and *Free* efficient, the class also needs a list of free registers—the `Stack` in `Class`. Routines *push* and *pop* manipulate the `Stack`.

With this level of detail, the code for both *Allocate* and *Free* is straightforward. Each class maintains 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, stores it, and reallocates the physical register for *vr*. Allocate sets the *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 *Next* field to its appropriate value after it finishes with the current operation. *Free* pushes the register onto the stack and resets its fields in the *class* structure.

The function *dist(vr)* returns the index in the block of the next reference to *vr*. The compiler can annotate each reference in the block with the appropriate value for *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

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

```
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)
```

FIGURE 13.2 Representing a Register Class in C

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.

Spill Dirty Value	Spill Clean Value
store x_2	
load x_3	load x_3 (overwriting x_1)
load x_2	load x_1

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.

Spill Clean Value	Spill Dirty Value
load x_3	
load x_1	store x_2
load x_3	load x_3
load x_1	load x_2

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 regis-

ter 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.

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.