Engineering a Compiler

Manuscript for the Third Edition (EaC 3^e)

Keith D. Cooper
Linda Torczon

13.3 LOCAL REGISTER ALLOCATION

Recall that a *basic block* is a maximal length sequence of straight-line code.

The simplest formulation of the register allocation problem is local allocation. Consider a single basic block and a single class of *k* physical registers. This problem captures many of the complexities of allocation and serves as a useful introduction to the concepts and terminology needed to discuss global allocation. To simplify the discussion, we will assume that one block constitutes the entire program.

The input code uses *source registers*, denoted $\texttt{sr}_i$.

The output code uses *physical registers*, denoted $\texttt{pr}_i$. The PRs correspond to target machine registers.

The input block contains a series of three-address operations, each of which has the form $\texttt{op}_i\ \texttt{sr}_i, \texttt{sr}_j \Rightarrow \texttt{sr}_m$. From a high-level view, the local register allocator rewrites the block to replace each reference to a source register (SR) with a reference to a specific physical register (PR). The allocator must preserve the input block's original meaning while it fits the computation into the *k* PRs provided by the target machine.

If, at any point in the block, the computation has more than *k* live values—that is, values that may be used in the future—then some of those values will need to reside in memory for some portion of their lifetimes. (*k* registers can hold at most *k* values.) Thus, the allocator must insert code into the block to move values between memory and registers as needed to ensure that all values are in PRs when needed and that no point in the code needs more than *k* PRs.

This section presents a version of Best's algorithm, which dates back to the original FORTRAN compiler. It is one of the strongest known local allocation algorithms. It makes two passes over the code. The first pass derives detailed knowledge about the definitions and uses of values; essentially, it computes LIVE information within the block. The second pass then performs the actual allocation.

Spill: When the allocator moves a live value from a PR to memory, it *spills* the value.

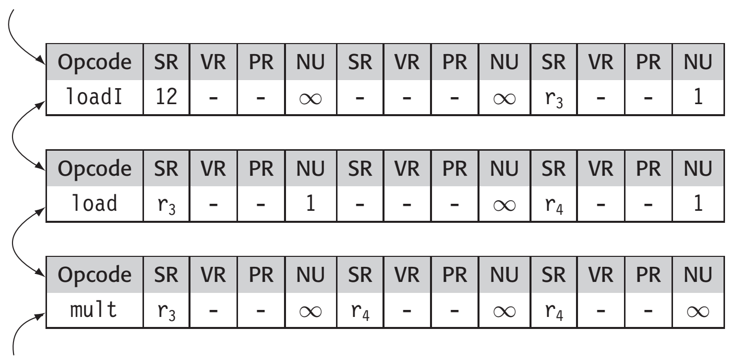
Restore: When the allocator retrieves a previously spilled value from memory, it *restores* the value.

Best's algorithm has one guiding principle: when the allocator needs a PR and they are all occupied, it should spill the PR that contains the value whose next use is farthest in the future. The intuition is clear; the algorithm chooses the PR that will reduce demand for PRs over the longest interval. If all values have the same cost to spill and restore, this choice is optimal. In practice, that assumption is rarely true, but Best's algorithm still does quite well.

To explain the algorithm it helps to have a concrete data structure. Assume a three-address, ILOC-like code, represented as a list of operations. Each operation, such as $\texttt{mult}\ \texttt{sr}_1, \texttt{sr}_2 \Rightarrow \texttt{sr}_3$ is represented with a structure:

	OPERAND 1				OPERAND 2				OPERAND 3			
Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
mult	r_1	-	-	∞	r_2	-	-	∞	r_3	-	-	∞

The operation has an opcode, one or two inputs (operands 1 and 2), and a result (operand 3). Each operand has a source-register name



Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
`loadI`	12	-	-	∞	-	-	-	∞	r_3	-	-	1

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
`load`	r_3	-	-	1	-	-	-	∞	r_4	-	-	1

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
`mult`	r_3	-	-	∞	r_4	-	-	∞	r_4	-	-	∞

FIGURE 13.3 Representing a List of Operations

(SR), a virtual-register name (VR), a physical-register name (PR), and the index of its next use (NU). The allocator's primary task is to manipulate the names associated with values; thus, keeping the SR, VR, and PR names separate simplifies both programming and debugging.

A list of operations might be represented as a doubly-linked list, as shown in Figure 13.3. The local allocator will need to traverse the list in both directions. The list could be created in an array of structure elements, or with individually-allocated or block-allocated structures.

The first operation, a `loadI`, has an immediate value as its first argument, stored in the SR field. It has no second argument. The next operation, a `load`, also has just one argument. The final operation, a `mult`, has two arguments. Because the code fragment does not contain a next use for any of the registers mentioned in the `mult` operation, their NU fields are set to ∞.

Since the meaning is clear, we store a `loadI`'s constant in its first operand's SR field.

13.3.1 Renaming in the Local Allocator

To simplify the local allocator's implementation, the compiler can first rename SRs so that they correspond to live ranges. In a single block, an LR consists of a single definition and one or more uses. The *span* of the LR is the interval in the block between its definition and its last use.

The renaming algorithm finds the live range of each value in a block. It assigns each LR a new name, its virtual register name (VR). Finally, it rewrites the code in terms of VRs. Renaming creates a one-to-one correspondence between LRs and VRs which, in turn, simplifies many of the data structures in the local allocator. The allocator then reasons about VRs, rather than arbitrary SR names.

The compiler can discover live ranges and rename them into VRs in a single backward pass over the block. As it does so, it can also collect and record next use information for each definition and use in the block. The algorithm, shown in Figure 13.4, assumes the representa-

```
VRName ← 0
for i ← 0 to max source-register number do
    SRToVR[i] ← invalid
    PrevUse[i] ← ∞

index ← block length
for each Op in the block, bottom to top, do
    for each operand, O, that OP defines do   // defs first
        if SRToVR[O.SR] = invalid then        // def has no uses
            SRToVR[O.SR] ← VRName++           // start a new VR anyway
        O.VR ← SRToVR[O.SR]                   // set VR and NU for O
        O.NU ← PrevUse[O.SR]
        PrevUse[O.SR] ← ∞
        SRToVR[O.SR] ← invalid                // next use of SR starts new VR
    for each operand, O, that OP uses do      // uses after defs
        if SRToVR[O.SR] = invalid then        // start a new VR
            SRToVR[O.SR] ← VRName++
        O.VR ← SRToVR[O.SR]                   // set VR and NU for O
        O.NU ← PrevUse[O.SR]
    for each operand, O, that OP uses do
        PrevUse[O.SR] ← index                 // save to set next NU
    index ← index - 1
```

FIGURE 13.4 Renaming Source Registers Into Live Ranges

tion described in the previous section.

The renaming algorithm builds two maps: *SRToVR*, which maps an SR name to a VR name, and *PrevUse*, which maps an SR name into the ordinal number of its most recent use. The algorithm begins by initializing each *SRToVR* entry to *invalid* and each *PrevUse* entry to ∞.

The algorithm walks the block from the last operation to the first operation. At each operation, it visits definitions and then uses. At each operand, it updates the maps and defines the *VR* and *NU* fields.

If the SR for a definition has no VR, that value is never used. The algorithm still assigns a VR to the SR.

When the algorithm visits a use or def, it first checks whether or not the reference's SR, *O.SR*, already has a VR. If not, it assigns the next available VR name to the SR and records that fact in *SRToVR[O.SR]*. Next, it records the VR name and next use information in the operand's record. If the operand is a use, it sets *PrevUse[O.SR]* to the current operation's index. For a definition, it sets *PrevUse[O.SR]* back to ∞.

Note that all operands to a `store` are uses. The `store` defines a memory location, not a register.

The algorithm visits definitions before uses to ensure that the maps are updated correctly in cases where an SR name appears as both a definition and a use. For example, in `add` r_{17}, $r_{18} \Rightarrow r_{18}$, the algorithm will rewrite the definition with *SRToVR*[r_{18}]; update *SRToVR*[r_{18}] with a new VR name for the use; and then set *PrevUse*[r_{18}] to ∞.

After renaming, each live range has a unique VR name. An SR name that is defined in multiple places will be rewritten as multiple distinct VR names. In addition, each operand in the block has its *NU* field set to either the ordinal number of the next operation in the block that uses its value, or ∞ if it has no next use. The allocator uses this information when it chooses which VRs to spill.

After renaming, we use *live range* and VR interchangeably.

Consider, again, the code from Figure 13.1. Panel (a) shows the original code. Panel (b) shows that code after renaming. Panel (c) shows the span of each live range, as an interval graph. The allocator does not rename r_{arp} because it is a dedicated PR that holds the activation record pointer and, thus, not under the allocator's control.

The maximum demand for registers, MAXLIVE, occurs at the start of the first `mult` operation, marked in panel (c) by the dashed gray line. Six VRs are live at that point. Both VR_7 and VR_8 are live at the start of the operation. VR_5's live range starts at the end of the operation, after VR_7's and VR_8's live ranges end.

MAXLIVE: The maximum number of concurrently live VRs in a block

13.3.2 Allocation and Assignment

The algorithm for the local allocator appears in Figure 13.5. It performs allocation and assignment in a single forward pass over the block. It starts with an assumption that no values are in physical registers. It iterates through the operations, in order, and incrementally allocates a PR to each VR. At each operation, the allocator performs three steps.

1. To ensure that a use has a PR, the allocator first looks for a PR in the *VRToPR* map. If the entry for VR is *invalid*, the algorithm calls *GetAPR* to find a PR. The allocator uses a simple marking scheme to avoid allocating the same PR to conflicting uses in a single operation.
2. In the second step, the allocator determines if any of the uses are the last use of the VR. If so, it can free the PR, which makes the PR available for reassignment, either to a result of the current operation or to some VR in a future operation.
3. In the third step, the allocator ensures that each VR defined by the operation has a PR allocated to hold its value. Again, the allocator uses *GetAPR* to find a suitable register.

If an instruction contains several operations, the allocator should handle all of the uses before any of the definitions.

Each of these steps is straightforward, except for picking the value to spill. Most of the complexity of local allocation falls in that task.

***The Workings of* GetAPR** As it processes an operation, the allocator will need to find a PR for any VR v that does not currently have one. This act is the essential act of register allocation. Two situations arise:

1. *Some PR p is free*: The allocator can assign p to v. The algorithm

```
for vr ← 0 to max VR number do
    VRToPR[vr] ← invalid

for pr ← 0 to max PR number do
    PRToVR[pr] ← invalid
    PRNU[pr] ← ∞
    push(pr)    // pop() appears in GetAPR()

// iterate over the block
for each Op in the block, in linear order, do
    clear the mark in each PR          // reset marks

    for each use, U, in OP do          // allocate uses
        pr ← VRToPR[U.VR]
        if (pr = invalid) then
            U.PR ← GetAPR(U.VR,U.NU)
            Restore(U.VR,U.PR)
        else
            U.PR ← pr
        set the mark in U.PR

    for each use, U, in OP do          // last use?
        if (U.NU = ∞ and PRToVR[U.PR] ≠ invalid) then
            FreeAPR(U.PR)

    clear the mark in each PR          // reset marks

    for each definition, D, in OP do   // allocate defs
        D.PR ← GetAPR(D.VR,D.NU)
        set the mark in D.PR
```

```
GetAPR(vr, nu)
    if stack is non-empty then
        x ← pop()
    else
        pick an unmarked x to spill
        Spill(x)

    VRToPR[vr] ← x
    PRToVR[x] ← vr
    PRNU[x] ← nu
    return x

FreeAPR(pr)
    VRToPR[PRToVR[pr]] ← invalid
    PRtoVR[pr] ← invalid
    PRNU[pr] ← ∞
    push(pr)
```

FIGURE 13.5 The Local Allocator

maintains a stack of free PRs for efficiency.

2. *No PR is free*: The allocator must choose a VR to evict from its PR p, save the value in p to its spill location, and assign p to hold v.

If the reference to v is a use, the allocator must then restore v's value from its memory location to p.

Best's heuristic states that the allocator should spill the PR whose current VR has the farthest next use. The algorithm maintains *PRNU* to facilitate this decision. It simply chooses the PR with the largest *PRNU*. If the allocator finds two PRs with the same *PRNU*, it must choose one.

The implementation of *PRNU* is a tradeoff between the efficiency of updates and the efficiency of searches. The algorithm updates *PRNU* at each register reference. It searches *PRNU* at each spill. The algorithm shows *PRNU* as a simple array; that reflects the assumption that updates are much more frequent than spills. If spills are frequent

Spill and Restore Code

At the point where the allocator inserts spill code, all of the physical registers (PRs) are in use. The compiler writer must ensure that the allocator can still spill a value.

Two scenarios are possible. Most likely, the target machine supports an address mode that allows the spill without need for an additional PR. For example, if the ARP has a dedicated register, say r_{arp}, and the ISA includes an address-immediate store operation, like ILOC's `storeAI`, then spill locations in the local data area can be reached without an additional PR.

On a target machine that only supports a simple load and store, or an implementation where spill locations cannot reside in the activation record, the compiler would need to reserve a PR for the address computation—reducing the pool of available PRs. Of course, the reserved register is only needed if MAXLIVE $> k$. (If MAXLIVE $\leq k$, then no spills are needed and the reserved register is also unneeded.)

enough, using a priority queue for *PRNU* may improve allocation time.

Tracking Physical and Virtual Registers To track the relationship between VRs and PRs, the allocator maintains two maps. *VRToPR* contains, for each VR, either the name of the PR to which it is currently assigned, or the value *invalid*. *PRToVR* contains, for each PR, either the name of the VR to which it is currently assigned, or the value *invalid*.

As it proceeds through the block, the allocator updates these two maps so that the following invariant always holds:

if *VRToPR[vr]* $\neq$ *invalid* then *PRToVR[VRToPR[vr]]* $= vr$.

The code in *GetAPR* and *FreeAPR* maintains these maps to ensure that the invariant holds true. In addition, these two routines maintain *PRNU*, which maps a PR into the ordinal number of the operation where it is next used—a proxy for distance to that next use.

Spills and Restores Conceptually, the implementation of *Spill* and *Restore* from Figure 13.5 can be quite simple.

- To spill a PR p, the allocator can use *PRToVR* to find the VR v that currently lives in p. If v does not yet have a spill location, the allocator assigns it one. Next, it emits an operation to store p into the spill location Finally, it updates the maps: *VRToPR*, *PRToVR*, and *PRNU*.
- To restore a VR v into a PR p, the allocator simply generates a load from v's spill location into p.

Spill locations typically are placed at the end of the local data area in the activation record.

If all spills have the same cost and all restores have the same cost, then

```
                                                 restore x3
spill   x2                        spill   x2     restore x1
restore x3     restore x3         restore x3     restore x3
restore x2     restore x1         restore x2     restore x1

Spill Dirty    Spill Clean        Spill Dirty    Spill Clean

   (a) References x3 x1 x2        (b) References x3 x1 x3 x1 x2
```

FIGURE 13.6 Spills of Clean versus Dirty Values

Best's algorithm generates an optimal allocation for a block.

Complications from Spill Costs In real programs, spill and restore costs are not uniform. Real code contains both clean and dirty values; the cost to spill a dirty value is greater than the cost to spill a clean value. To see this, consider running the local allocator on a block that contains references to three names, x_1, x_2, and x_3, with just two physical registers ($k = 2$).

Assume that the register allocator is at a point where x_1 and x_2 are currently in registers and x_1 is clean and x_2 is dirty. Figure 13.6 shows how different spill choices affect the code in two different scenarios.

Reference string: A reference string is just a list of references to registers or addresses. In this context, each reference is a use, not a definition.

Panel (a) considers the case when the reference string for the rest of the block is $x_3\ x_1\ x_2$. If the allocator consistently spills clean values before dirty values, it introduces less spill code for this reference string.

Panel (b) considers the case when the reference string for the rest of the block is $x_3\ x_1\ x_3\ x_1\ x_2$. Here, if the allocator consistently spills clean values before dirty values, it introduces more spill code.

The presence of both clean and dirty values fundamentally changes the local allocation problem. Once the allocator faces two kinds of values with different spill costs, the problem becomes NP-hard. The introduction of rematerializable values, which makes restore costs non-uniform, makes the problem even more complex. Thus, a fast deterministic allocator cannot always make optimal spill decisions. The local allocator still produces good local allocations.

Remember, however, that the problem is NP-hard. No efficient, deterministic algorithm will always produce optimal results.

In practice, the allocator may produce better allocations if it differentiates between dirty, clean, and rematerializable values (see Section 13.2.3). If two PRs have the same distance to their next uses and different spill costs, then the allocator should spill the lower-cost PR.

The issue becomes more complex, however, in choosing between PRs with different spill costs that have next-use distances that are close but not identical. For example, given a dirty value with next use of n and a rematerializable value with next use of $n - 1$, the latter value will sometimes be the better choice.

SECTION REVIEW

The limited context in local register allocation simplifies the problem enough so that a fast, intuitive algorithm can produce high-quality allocations. The local allocator described in this section operates on a simple principle: *when a PR is needed, spill the PR whose next use is farthest in the future*.

In a block where all values had the same spill costs, the local allocator would achieve optimal results. When the allocator must contend with both dirty and clean values, the problem becomes combinatorially hard. A local allocator can produce good results, but it cannot guarantee optimal results.

REVIEW QUESTIONS

1. Modify the renaming algorithm, shown in Figure 13.4, so that is also computes MAXLIVE, the maximum number of simultaneously live values at any instruction in the block.
2. Rematerializing a known constant is an easy decision, because the spill requires no code and the restore is a single load immediate operation. Under what conditions could the allocator profitably rematerialize an operation such as `add` $r_a, r_b \Rightarrow r_x$?