Algorithm in Figure 13.1 corrected per Piazza post.

Engineering a Compiler

Manuscript for the Third Edition (EaC 3e)

Keith D. Cooper
Linda Torczon

Rice University
Houston, Texas

Limited Copies Distributed
Reproduction requires explicit written permission

Copyright 2018, Morgan-Kaufmann Publishers and the authors
All rights reserved
Chapter 13

Register Allocation

13.1 LOCAL ALLOCATION AND ASSIGNMENT

As an introduction to the issues that arise in register allocation, consider the simplest formulation of the problem: allocation for a single basic block with a single class of \( k \) physical registers. The local allocation 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 block contains a series of three-address operations, each of which has the form \( \text{op}_i, \text{sr}_i, \text{sr}_j \Rightarrow \text{sr}_k \). 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, value 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 cannot hold more than \( k \) 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 derives detailed knowledge about the definitions and uses of values, and the second performs both allocation and assignment.

At a high-level of abstraction, 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 create a register allocator from this intuition requires much more detail. Sections 13.1.1 and 13.1.2 present the key algorithms of the local allocator. Section 13.1.3 discusses how to extend the algorithm to handle machines with overlapping register classes.

---

**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.
To explain the algorithms, it helps to have a concrete data structure. We will assume a three-address, ILOC-like code, represented as a list of operations. An operation, such as \texttt{mul\ sr}_1, \texttt{sr}_2 \Rightarrow \texttt{sr}_3 will be represented as:

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\textbf{Opcode} & \textbf{OP1} & \textbf{OP2} & \textbf{OP3} \\
\hline
\texttt{mul} & \texttt{sr}_1 & \texttt{sr}_2 & \texttt{sr}_3 \\
\hline
\end{tabular}
\end{center}

The operation has an opcode, one or two inputs (\texttt{OP1} and \texttt{OP2}), and a result (\texttt{OP3}). Each operand, input or result, has a source-register name (SR), a virtual-register name (VR), a physical-register name (PR), and a distance to next use (NU). The real job of a register allocator is to manipulate the names associated with values; thus, keeping the source, virtual, and physical register names separate will simplify both programming and debugging.

A list of operations might be represented as a doubly-linked list. The local allocator will need to traverse the list in both directions.

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

The first operation, a \texttt{loadI} has an immediate value as its first argument, stored in the SR field. It has no second argument. The second operation, a \texttt{load} also has just one argument. The third operation, a \texttt{mul} has two arguments. The NU fields are set to show the relative distance to the value’s next use. Because the code fragment does not contain a next use for any of the registers mentioned in the \texttt{mul} operation, their NU fields are set to \(\infty\).

### 13.1.1 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, a live range consists of a single definition and one or more uses. The span of the live range is the interval in the block between its definition and its last use. (The notion of a live range becomes more complex in the presence of control flow.)
13.1 Local Allocation and Assignment

VRName ← 0
for i ← 0 to max source-register number
    SRToVR[i] ← invalid
    LastUse[i] ← ∞
index ← block length
for each Op in the block, bottom to top
    for each operand, O, that OP defines // 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 ← LastUse[O.SR]
            LastUse[O.SR] ← ∞
            SRToVR[O.SR] ← invalid // next use of SR starts new VR
    for each operand, O, that OP uses // 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 ← LastUse[O.SR]
            LastUse[O.SR] ← index // save to set next NU
    index ← index - 1

**Figure 13.1** Renaming Source Registers Into Live Ranges

The renaming algorithm finds the live range of each value; assigns each live range a new name: its virtual register name (VR); and rewrites the code in terms of VRs. Renaming creates a one-to-one correspondence between live ranges and VRs which, in turn, simplifies many of the data structures in the local allocator. The allocator then reasons about live ranges, 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 13.1, assumes the representation described in the previous section.

The renaming algorithm builds two maps: `SRToVR` and `LastUse`. The first, `SRToVR`, maps an SR name to a VR name. The second, `LastUse`, maps an SR name into an ordinal number for the operation in which it was last used. The algorithm begins by initializing each `SRToVR` slot to `invalid` and each `LastUse` slot to `∞`.

The algorithm walks the block from the last operation to the first operation. At each operation, it updates the maps and defines the `VR` and `NU` fields for each operand. Within an operation, it visits operands and
that are defined before operands that are used.

When the algorithm visits a use or def, it first checks whether or not the reference's SR, \textit{O.SR}, already has a VR. If not, it assigns the next available VR name to the SR and records that fact in \textit{SRToVR[O.SR]}. Next, it records the VR name and next use information in the operand's record. Finally, it sets \textit{LastUse[O.SR]} to the current operation's index.

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 \texttt{add}, \texttt{r17 \Rightarrow r18}, the algorithm will rewrite the definition with \textit{SRToVR[r18]}, update \textit{SRToVR[r18]} with a new VR name for the use; and then set \textit{LastUse[r18]} to $\infty$.

After renaming, each live range has a unique VR number. An SR name that is defined in multiple places will be rewritten as multiple distinct VR names. In addition, each register 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 $\infty$ if it has no next use. The allocator uses this particular value to choose which VRs to spill.

Figure 13.2 shows the example block from Section 1.3.3 on page 17 before and after renaming. The example assumes that \texttt{r_{arp}} is a reserved PR that holds the activation record pointer (ARP). \texttt{r_{arp}} is not renamed. Panel (c) shows the span of each live range, as an interval graph. Within an operation, the uses appear slightly above the definitions.

The maximum demand for registers, \texttt{MAXLIVE}, occurs at the start of the first \texttt{mult} operation, marked by the dashed gray line. Six VRs are live at that point. Both \texttt{vr_7} and \texttt{vr_8} are live at the start of the operation. \texttt{vr_5}'s live range starts at the end of the operation, after \texttt{vr_7}'s and \texttt{vr_8}'s live ranges end.
13.1 Local Allocation and Assignment

for vr ← 0 to max VR number
  VRTOPR[vr] ← invalid

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

// iterate over the block
for each Op in the block, in linear order
  // iterate over operands in OP
  for each use, U, in OP // allocate uses
    if (U.PR is invalid) then
      U.PR ← GetAPR(U.VR, U.NU)
      Restore(U.VR, U.PR)

  for each use, U, in OPi // last use?
    if (U.NU = ∞) then
      FreeAPR(U.PR)
      push(pr)

for each definition, D, in OP // allocate defs
  D.PR ← GetAPR(D.VR, D.NU,i)

GetAPR(vr, nu)
  if stack is non-empty then
    x ← pop()
  else
    pick an x to spill
    Spill(x)
  VRTOPR[vr] ← x
  PRTOPR[x] ← vr
  PRNU[x] ← nu
  return x

FreeAPR(pr)
  VRTOPR[PRTOPR[pr]] ← invalid
  PRTOPR[pr] ← invalid
  PRNU[pr] ← ∞

Figure 13.3 The Local Allocator

13.1.2 Allocation and Assignment

The local allocator performs allocation and assignment in a single forward pass over the basic block as shown in Figure 13.3. It starts with an assumption that no values are in physical registers. It iterates through the operations and incrementally allocates a PR to each VR.

At each operation, the allocator performs three steps. First, it checks each use to determine if the VR already has an assigned PR. If it does not, it finds a PR $x$ that is not already used in the operation, assigns $x$ to the VR, and emits code to load the VR’s value into $x$.

After all of the uses in an operation have been assigned a PR, the allocator re-checks the uses to determine if any of them are the last use of the VR. If so, it can free the PR, which makes the PR available for reassignment, either to the result of the current operation or to a use or definition in a future operation. At that point, the allocator can find and assign PRs for definitions in the operation.

Each of these steps are straight-forward, except for the step “pick an x to spill” in GetAPR. Most of the complexities of local register allocation are folded into this single task.

Tracking Physical and Virtual Registers To track the relationship between VRs and PRs, the allocator maintains two maps. VRTOPR con-
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. The first, and more likely, situation is that 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 ≤ k, then no spills are needed and the reserved register is also unneeded.)

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

\[
\text{if } VRToPR[vr] \neq \text{invalid} \text{ then } PRToVR[VRToPR[vr]] = vr.
\]

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.

Finding a Physical Register

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 is free:** The allocator can assign a free PR p to v. The algorithm 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 searches. At each operation, each register reference updates $PRNU$. It is searched each time the allocator must choose a PR to 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 enough, using priority queue for $PRNU$ may improve allocation time.

**Spills and Restores** Conceptually, the implementation of Spill and Restore from Figure 13.3 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 generates a store operation from $p$ to the spill location. Finally, it updates the three 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 $p$. It then updates the three maps, $VRToPR$, $PRToVR$, and $PRNU$.

If every spilled value requires a store—that is, the cost of spilling is identical for all values—and every restored value requires a load—that is, the cost of restoring is identical for all values—then Best’s algorithm produces an optimal allocation for straight-line code.

In practice, different values have different costs to spill and restore. Consider three different situations for a VR $v$:

- **$v$ is inexpensive to re-create:** Some values can be re-created in fewer cycles that it takes to spill and restore. Such a value can be discarded without a spill and re-created before its next use. We call these values rematerializable values.

  A common source of rematerializable values is an immediate load operation. Rather than using a store to spill and a load to restore, the allocator can re-create the value in a PR by inserting an (inexpensive) immediate load operation before its next use.

- **$v$ is clean:** If $v$ exists in both a PR and in memory, it does not require a spill before the PR can be reused. Instead, the allocator can restore $v$ from the copy of its value in memory.

  Clean values arise in two ways. The value $v$ may have already been spilled, in which case a copy of $v$ exists in $v$’s spill location. The value $v$ may be the result of a load operation, in which case a copy of $v$ exists in $v$’s primary memory location.
Complications from Spill Costs

The presence of both clean and dirty values complicates the choice of a value to spill. Consider a block with references to $x_1$, $x_2$, and $x_3$, at a point where $x_1$ and $x_2$ are in PRs. The allocator knows that $k = 2$, $x_1$ is clean, and $x_2$ is dirty.

Consider what happens if the reference string for the rest of the block is $x_3$, $x_1$, $x_2$. Panel (a) below compares the sequence of spills and restores produced if the allocator always spills the dirty value with the code produced if it always spills the clean value. For this reference string, spilling the clean value, $x_1$, yields fewer spills and restores than spilling the dirty value, $x_2$, even though $x_1$’s next use is before $x_2$’s next use.

<table>
<thead>
<tr>
<th>Spill</th>
<th>Restore</th>
<th>Spill</th>
<th>Restore</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_2$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$x_3$</td>
<td>$x_2$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$x_1$</td>
<td>$x_3$</td>
<td>$x_3$</td>
</tr>
<tr>
<td>Spill Dirty</td>
<td>Spill Clean</td>
<td>Spill Dirty</td>
<td>Spill Clean</td>
</tr>
</tbody>
</table>

Panel (b) shows the sequence of spills generated for the reference string $x_3$, $x_1$, $x_3$, $x_1$, $x_2$, under the same initial conditions. Here, the better choice is to spill the dirty value.

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, local allocation becomes NP-hard. In practice, that simply means that a fast deterministic allocator will not always make optimal spill decisions. The local allocator still produces good local allocations.

$v$ is dirty: If $v$ only exists in a PR $p$, it must be spilled to memory before $p$ can be used for another value. Thus, choosing $p$ incurs the full cost of a spill, as well as a restore before the next use.

Typically, a dirty value is the result of a computation that is still live and has not yet been spilled. An ambiguous store can force the allocator to treat some clean values as dirty values.

The allocator can sometimes produce better allocations by considering the actual cost of spills and restores, rather than assuming all values have a uniform cost to spill and to restore. This approach complicates the selection of a PR to spill in $GetAPR$, but it can lead to reduced overhead from spills and restores. 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 often be the better choice.

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