Engineering a Compiler

Manuscript for the Third Edition (EaC3e)

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
13.2 **BACKGROUND**

The design and implementation of a register allocator is a complex task. The decisions made in allocator design may depend on decisions made earlier in the compiler. At the same time, design decisions for the allocator may dictate decisions in earlier passes of the compiler. This section introduces some of the issues that arise in allocator design.

### 13.2.1 A Name Space for Allocation: Live Ranges

At its heart, the register allocator creates a new name space for the code. It receives code written in terms of virtual registers and memory locations; it rewrites the code in a way that maps those VRs onto both the physical registers and some additional memory locations.

To improve the efficiency of the generated code, the allocator should minimize unneeded data movement, both between registers and memory, and among registers. If the allocator decides to keep some value \( x \) in a register, it should arrange, if possible, for each definition of \( x \) to target the same register and for each use of \( x \) to read that register. This goal, if achieved, eliminates unneeded register-to-register copy operations; it may also eliminate some stores and loads.

This goal suggests a name space that assigns a unique name to these connected webs of definitions and uses. The register allocator analyzes the flow of values in the code, constructs a set of live ranges, and renames the code so that it uses these names. A live range (LR) consists of all the definitions that reach some use, plus all of the uses that those definitions reach. This collection of definitions and uses forms a closed set.

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

A live range corresponds to a single value and consists of all the operations where that value is live.
If we confine our attention to a single block, live ranges are easy to visualize and understand. Each LR corresponds to a single definition and extends from that definition to the value’s last use. Figure 13.1.a shows an ILOC fragment that appeared in Chapter; panel (b) shows the code renamed into its distinct live ranges. The live ranges are shown as a graph in panel (c). The graph can be summarized as a set of intervals; for example LR2 is [6,10] and LR8 is [3,7]. An interval begins immediately after the first operation defines its target register and ends when the last operation reads its operands. The drawing in panel (c) assumes no overlap between execution of the operations.

The last use of an LR occurs for one of two reasons. The operation may be the last use of that name along the current path through the code. Alternatively, the name might be redefined before its next use, which starts a new LR.

In a CFG with control flow, the situation is more complex, as shown in the margin. Consider the variable x. Its three definitions form two separate and distinct live ranges.

1. The use in B4 refers to two definitions: the one in B1 and the one at the bottom of B2. These three events create the first LR, denoted LR1. LR1 spans B1, B3, B4, and the last statement in B2.

2. The use of x in B2 refers only to the definition immediately above it in B2. This pair creates a second LR, denoted LR2. LR1 and LR2 are independent of each other.

With more complex control flow, the shapes of live ranges in the CFG can become more complex. The interval notation, which works well in a single block, does not capture the complexity of this situation.

**Variations on Live Ranges** Different allocation algorithms have defined *live range* in distinct ways. The local allocator described in Section 13.3 treats the entire lifetime of a value in the basic block as a single live range; it uses a maximal-length live range within the block. The global allocator described in Section 13.4 similarly uses a maximal-length live range within the entire procedure.

Other allocators use different notions of a live range. The linear scan allocators use an approximation of live ranges that overestimates their length but produces an interval representation that leads to a faster allocator. The SSA-based allocators treat each SSA name as a separate live range; it must then translate out of SSA form after allocation. Several allocators have restricted the length of live ranges so that they conform to features in the control-flow graph, such as loops, so that spill code tends to occur before and after loops, rather than inside them. Section 13.5.3 describes some of these other formulations.

**Code Shape and Live Ranges** The register allocator must understand when a source-code variable can legally reside in a register. If
the variable is ambiguous, it can only reside in a register between its creation and the next store operation in the code (see Section 4.6.2). If it is unambiguous and scalar—that is, a single value that can fit in a register—then the allocator can choose to keep it in a register over a longer period of time.

The compiler has two major ways to determine when a value is unambiguous. It can perform static analysis to determine which values are safe to keep in a register; such analysis can range from inexpensive and imprecise through costly and precise. Alternatively, it can encode knowledge of ambiguity into the shape of the code.

If the compiler uses a register-to-register memory model, it can allocate a virtual register to each unambiguous value. If the VR is live after the defining procedure’s return, as with a static value or a call-by-reference parameter, it will also need a memory home. The compiler can save the VR at the necessary points in the code.

If the compiler uses a memory-to-memory model, the allocator will still benefit from knowledge about ambiguity. The compiler should record that information with the symbol table entry for each value.

13.2.2 Interference

The register allocator’s most basic function is to determine, for two live ranges or values, whether or not they can occupy the same register. Two LRs can share a register if they use the same class of register and they are not simultaneously live. If two LRs use the same class of register, but there exists an operation where both LRs are live, then those LRs cannot use the same register, unless the compiler can prove that they have the same value. We say that such live ranges interfere.

Two LRs that use different classes of registers cannot interfere with each other because they do not compete for the same resource. Thus, for example, a floating point LR cannot interfere with an integer LR on a processor that uses distinct registers for these two kinds of values.

In the example from the margin, the two LRs for x do not interfere; LR2 is only live inside B2, in a stretch of code where LR1 is dead. Thus, the allocation decisions for LR1 and LR2 are independent. They could share a register, but there is no inherent reason for that to happen.

Global allocators operate by finding interferences and using them to guide the allocation process. The allocator described in Section 13.4 builds a concrete representation of these conflicts, an interference graph, and constructs a coloring of the graph to map live ranges onto physical registers. Many global allocators follow this paradigm; they vary in the graph’s precision and the specific coloring algorithm used.

Finding Interferences To discover interferences, the compiler first computes live information for the code. Then, it considers each opera-
tion in the scope being allocated. At each operation, the value defined by the operation interferes with every other live value.

The one exception to this rule is a copy operation, which sets the value of the LR that it defines to the value of the LR that it uses. Because the source and destination LRs have the same value, the copy operation does not create an interference between them.

**Interference and Register Pressure**  The interference graph provides a quick way to estimate the demand for registers, often called register pressure. For a node LR$_i$ in the graph, the degree of LR$_i$, written LR$^\circ_i$, is the number of neighbors that LR$_i$ has in the graph. If all of LR$_i$’s neighbors are live at the same operation, then LR$^\circ_i + 1$ registers would be needed to keep all of these values in registers. If those values are not all live at the same operation, then the register pressure may be lower than the degree. Maximum degree across all the nodes in the interference graph provides a quick upper bound on the number of registers required to execute the program without any spilling.

**Representing Physical Registers**  Often, the allocator will include nodes to represent physical registers in the interference graph. These nodes allow the compiler to specify both connections to PRs and interferences with PRs. For example, if the code passes LR$_i$ as the second parameter at a call site, the compiler could record that fact with a copy from LR$_i$ to the PR that will hold the second parameter.

Some compilers use PRs to control assignment of an LR. To force LR$_i$ into PR$_j$, the compiler can add a pseudo-interference from LR$_i$ to every PR except PR$_j$. Similarly, to prevent LR$_i$ from using PR$_j$, the compiler can add an interference between LR$_i$ and PR$_j$. While this mechanism works, it can become cumbersome. The mechanism for handling overlapping register classes presented in Section 13.4.7 provides a more general and elegant way to control placement in a specific PR.

### 13.2.3 Spill Code

When the allocator decides that it cannot keep some LR in a register, it must spill that LR to memory before re-allocating its PR to another value. It must also restore the spilled LR into a PR before any subsequent use. These added spills and restores increase execution time, so the allocator should insert as few of them as practical. The most direct measure of allocation quality is the time spent in spill code at runtime.

Allocators differ in the granularity with which they spill values. The global allocator described in Section 13.4 spills the entire live range. When it decides to spill LR$_i$, it inserts a spill after each definition in LR$_i$ and a restore immediately before each use of LR$_i$. In effect, it breaks LR$_i$ into a set of tiny LRs, one at each definition and each use.
By contrast, the local allocator described in Section 13.3 spills a live range only between the point where its PR is reallocated and its next use. Because it operates in a single block, with straight-line control flow, it can easily implement this policy; the LR has a unique next use and the point of spill always precedes that use.

Between these two policies, “spill everywhere” and “spill once”, lie many possible alternatives. Researchers have experimented with spilling partial live ranges. The problem of selecting an optimal granularity for spilling is, undoubtedly, as hard as finding an optimal allocation; the correct granularity likely differs between live ranges. Section 13.5 describes some of the schemes that have been tried.

**Non-uniform Spill Costs** To further complicate spilling, the allocator should account for the fact that properties of a LR can change the cost to spill it and to restore it.

**Dirty value** In the general case, the LR contains a value that has been computed and has not yet been stored to memory; we say that the LR is dirty. A dirty LR requires a store at its spill points and a load at its restore points.

**Clean value** If the value of the LR already exists in memory, then its spill points do not require stores; we say that the LR is clean. A clean LR costs nothing to spill; its restores cost the same as those of a dirty value.

**Rematerializable value** Some LRs contain values that cost less to recompute than to spill and restore. If the values used to compute the LR’s value are available at each use of the LR, the allocator can simply recompute it on demand. The classic example is an LR defined by an immediate load. Such an LR costs nothing to spill; to restore it, the compiler inserts the recomputation. Typically, an immediate load is cheaper than a load from memory.

The allocator should, to the extent possible, account for the non-uniform nature of spill costs. Of course, doing so complicates the allocator. Furthermore, the NP-complete nature of allocation suggests that no simple heuristic will make the best choice in every situation.

**Spill Locations** When the allocator spills a dirty value, it must place the value somewhere in memory. If the LR corresponds precisely to a variable kept in memory the allocator could spill the value back to that location. Otherwise, it must reserve a location in memory to hold the value during the time that the value is live.

Most allocators place such spill locations at the end of the procedure’s local data area. This decision allows the spill and restore code to access the value at a fixed offset from the ARP, making use of an address-immediate memory operation if the target machine provides
one. The allocator simply increases the size of the local data area, at compile time, so the allocation incurs no direct runtime cost.

Because an LR is only of interest during that portion of the code where it is live, the allocator has the opportunity to reduce the amount of spill memory that it uses. If LR$_i$ and LR$_j$ do not interfere, they can share the same spill location. Thus, the allocator can also use the interference graph to color spill locations and reduce memory use for spills.

### 13.2.4 Register Classes

Modern processors support multiple classes of registers. For example, most processors have one set of general purpose registers (GPRs) for use in integer operations and address manipulation, and another set of floating point registers (FPRs). In the case of GPRs and FPRs, the two register classes are, almost always, implemented with physically and logically disjoint register sets.

Most processors overlay multiple register classes onto a single physical register set. As shown in Figure 13.2.a, the ARM A-64 supports four sizes of floating point values in one set of quad-precision (128 bit) FPRs. The 128 bit FPRs are named Q0, Q1, ..., Q31. Each Qi is overlaid with a 64 bit register Di, a 32 bit register Si, and a 16 bit register Hi. The shorter registers occupy the low-order bits of the longer registers.

The ARM A-64 GPRs follow a similar scheme. The 64-bit GPRs have both 64-bit names Xi and 32-bit names Wi. Again, the 32-bit field occupies the low order bits of the 64-bit register.

The Intel IA-32 has a small register set, part of which is depicted in Figure 13.2.b. It provides eight 32-bit registers for program use. The CISC instruction set uses distinct registers for specific purposes, lead-
ing to unique names for each register, as shown. For backward compatibility with earlier 16-bit processors, the physical register set supports naming subfields of the 32-bit registers.

- In four of the registers, the programmer can name the 32-bit register, its lower 16 bits, and two 8-bit fields. These registers are the accumulator (EAX), the count register (ECX), the data register (EDX), and the base register (EBX).

- In the other four registers, the programmer can name both the 32-bit register and its lower 16 bits. These registers are the base of stack (EBP), the stack pointer (ESP), the string source index pointer (ESI), and the string destination index pointer (EDI).

The figure omits the instruction pointer (EIP and IP) and flag register (EFLAGS and FLAGS) which have both 32-bit and 16-names. The later IA-64 features a larger set of 32-bit general purpose registers, but preserves the IA-32 names and features in the low numbered registers.

Many earlier architectures used pairing schemes in the floating point register set. The drawing in the margin shows how a four register set might work. It would consist of the four 32 bit physical registers, F0, F1, F2, and F3. 64 bit values occupy a pair of adjacent registers. If a register pair can begin with any register, then four pairs are possible: D0, D1, D2, and D3.

Some architectures restrict a register-pair to begin with an odd-numbered register—an aligned pair. With aligned pairs, only the registers shown as D0 and D2 would be available. With aligned pairs, use of D0 precludes the use of two 32-bit registers, F0 and F1, and only one 64 bit register, itself. With unaligned pairs, use of D0 precludes the use of the same two 32-bit registers. However, it precludes the use of three 64 bit registers: itself, D1, and D3.

In general, the register allocator should make effective use of all available registers. Thus, it must understand the processor’s register classes and include mechanisms to use them in a fair and efficient manner. For physically disjoint classes, such as floating point and general purpose register classes, the allocator can simply allocate them independently. Because floating point allocation spills typically make use of general purpose registers, the compiler should allocate floating point registers before general purpose registers.

Overlapping register classes require more careful attention; the compiler has complete responsibility for correct use of overlapping register classes. The ARM A-64 naming scheme allows the allocator to treat all of the fields in a single physical register as a single resource; it can use one of X0 or W0. By contrast, the IA-32 allows concurrent use of both AH and AL. The allocator needs more complex mechanisms to deal with this situation. Section 13.4.7 describes one way to build these mechanisms into a global graph-coloring register allocator.
13.3 LOCAL REGISTER ALLOCATION

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 block contains a series of three-address operations, each of which has the form \( op_i \ sr_i, sr_j \Rightarrow sr_k \). 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, 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.

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. We will assume a three-address, ILOC-like code, represented as a list of operations. An operation, such as \( \text{mult} \ sr_1, sr_2 \Rightarrow sr_3 \), will be represented with a structure:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>OPERAND 1</th>
<th>OPERAND 2</th>
<th>OPERAND 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OP1</td>
<td>OP2</td>
<td>OP3</td>
</tr>
<tr>
<td>mult</td>
<td>( r_1 )</td>
<td>( r_2 )</td>
<td>( r_3 )</td>
</tr>
</tbody>
</table>

The operation has an opcode, one or two inputs (OP1 and OP2), and a result (OP3). Each operand, input or result, has a source-register name (SR), a virtual-register name (VR), a physical-register name (PR), and

The input code uses source registers, denoted \( sr_i \).
The output code uses physical registers, denoted \( pr_i \). The PRs correspond to target machine registers.
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, 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 second operation, a load also has just one argument. The third operation, a mult 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 mult operation, their NU fields are set to $\infty$.

### 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. It assigns each LR a new name, its virtual register name (VR). Finally, it rewriting 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.
VRName ← 0
for i ← 0 to max source-register number
   SRToVR[i] ← invalid
   PrevUse[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 ← PrevUse[O.SR]
         PrevUse[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 ← PrevUse[O.SR]
         PrevUse[O.SR] ← index // save to set next NU
   index ← index - 1

**Figure 13.4** Renaming Source Registers Into Live Ranges

in the block. If the algorithm encounters a definition that has no uses, it will assign a VR to the definition, even though it has no real “range.” The algorithm, shown in Figure 13.4, assumes the representation 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.

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 ∞.

The algorithm visits definitions before uses to ensure that the maps are updated correctly in cases where an SR name appears as both a

If the SR for a def has no VR, that value is never used. The SR still receives a VR.

Note that all operands to a store are uses. The store defines a memory location, not a register.
definition and a use. For example, in add \( r_{17} \rightarrow r_{18} \), 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 \( \text{PrevUse}[r_{18}] \) 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 information to choose which VRs to spill.

Returning to Figure 13.1, the example assumes that \( r_{\text{arp}} \) is a dedicated PR that holds the activation record pointer. \( r_{\text{arp}} \) is not renamed. Panel (c) shows the span of each live range, as an interval graph.

The maximum demand for registers, \( \text{MAXLIVE} \), occurs at the start of the first \( \text{mul} \) 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.

### 13.3.2 Allocation and Assignment

The local allocator performs allocation and assignment in a single forward pass over the basic block as shown in Figure 13.5. 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 \( \text{VRToPR} \) map. If the entry for VR is \( \text{invalid} \), the algorithm calls \( \text{GetAPR} \) to find a PR. The allocator uses a 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 \( \text{GetAPR} \) to find a suitable register.

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 \( \text{GetAPR} \)** 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:
for vr ← 0 to max VR number
    VRToPR[vr] ← invalid
for pr ← 0 to max PR number
    PRToVR[pr] ← invalid
    PRNU[pr] ← ∞
    push(pr) // pop() appears in GetAVR()

// iterate over the block
for each Op in the block, in linear order
    clear the mark in each PR // reset marks
for each use, U, in OP // 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 // last use?
    if (U.NU = ∞ and U.PR ≠ invalid) then
        FreeAPR(U.PR)
        push(U.PR)
    clear the mark in each PR // reset marks
for each definition, D, in OP // allocate def
    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] ← ∞

\textbf{Figure 13.5} The Local Allocator

1. \textit{Some PR is free}: The allocator can assign \( p \) to \( v \). The algorithm maintains a stack of free PRs for efficiency.

2. \textit{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 \textit{PRNU} to facilitate this decision. It simply chooses the PR with the largest \textit{PRNU}. If the allocator finds two PRs with the same \textit{PRNU}, it must choose one.

The implementation of \textit{PRNU} is a tradeoff between the efficiency of updates and the efficiency of searches. The algorithm updates \textit{PRNU} at each register reference. It searches \textit{PRNU} at each spill. The algo-
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 \( \text{MAXLIVE} > k \). (If \( \text{MAXLIVE} \leq k \), then no spills are needed and the reserved register is also unneeded.)

The algorithm shows \( \text{PRNU} \) as a simple array; that reflects the assumption that updates are much more frequent than spills. If spills are frequent enough, using a priority queue for \( \text{PRNU} \) may improve allocation time.

Tracking Physical and Virtual Registers

To track the relationship between VRs and PRs, the allocator maintains two maps. \( \text{VRToPR} \) contains, for each VR, either the name of the PR to which it is currently assigned, or the value \( \text{invalid} \). \( \text{PRToVR} \) contains, for each PR, either the name of the VR to which it is currently assigned, or the value \( \text{invalid} \).

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

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

The code in \textit{GetAPR} and \textit{FreeAPR} maintains these maps to ensure that the invariant holds true. In addition, these two routines maintain \( \text{PRNU} \), 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 \textit{Spill} and \textit{Restore} from Figure 13.5 can be quite simple.

- To spill a PR \( p \), the allocator can use \( \text{PRToVR} \) 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, \( \text{VRToPR} \), \( \text{PRToVR} \), and \( \text{PRNU} \).
13.3 Local Register Allocation

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.

**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 $x_3 x_1 x_2$. Figure 13.6 shows how the choice between dirty and clean values affects the resulting code. For the reference string $x_3 x_1 x_2$, in panel (a), choosing the clean value yields fewer spills and restores. For the reference string $x_3 x_1 x_3 x_1 x_2$, in panel (b), choosing the dirty value yields fewer spills and restores.

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 means that a fast deterministic allocator will not always make optimal spill decisions. The local allocator still produces good local allocations.

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

The issue becomes more complex, however, in choosing between LRs 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 often be the better choice.

**Figure 13.6** Spills of Clean versus Dirty Values

- 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 $PRNU$.

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.

Remember, however, that the problem is NP-hard. No efficient, deterministic algorithm will always produce optimal results.
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 it also computes MAXLIVE, the maximum number of simultaneously live values at any instruction in the block.
2. Consider a target ISA that has no address-immediate mode on memory operations. Under what circumstances will the allocator need to reserve an extra register for the address computation on a spill or a restore?