8. Register Allocation

8.1 The Register Allocation Problem

Register allocation is the process of assigning as many local variables and temporaries to physical registers as possible. The more values that we can keep in registers instead of in memory, the faster our programs will run. This makes register allocation the most effective optimization technique that we have.

With respect to our LIR discussed in Chapter 7, we wish to assign physical registers to each of the virtual registers that serve as operands to instructions. The problem is that there are often many fewer physical registers than there are virtual registers. In this case, as program execution progresses, some values in physical registers will have to be *spilled* to memory while the register is used for another purpose, and then reloaded when those values are needed again. Code must be generated for storing spilled values and then for reloading those values at appropriate places. Of course, we wish to minimize this spilling (and reloading) of values to (and from) memory.

So, any register allocation strategy must determine how to most effectively allocate physical registers to virtual registers and, when spilling is necessary, *which* physical registers to spill to make room for assignment to other virtual registers. This problem has been shown to be NP-complete in general (Sethi, 1973) but there are several allocation strategies which do a reasonable job in near-linear time.

Register allocation that focuses on just a single basic block, or even just a single statement, is said to be *local*. Register allocation that considers the entire flow graph of a method is said to be *global*.

In this chapter we look briefly at some local register allocation strategies but we focus most of our attention on two global register allocation strategies: *linear scan register allocation* and *graph coloring register allocation*.

8.2 Naïve Register Allocation

A naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning global registers to virtual registers. Once all physical registers have been assigned, and if there are additional virtual registers to deal with, we begin spilling physical registers to memory. There is no strategy to determining which registers to spill; for example, one might simply sequence through the physical registers a second time in the same order they were assigned the first time, spilling each to memory as it is re-needed. When a spilled value used again, it must be reloaded into a (possibly different) register. Such a regime works just fine when there are as many physical registers as there are virtual registers; in fact, it is as effective as any other register allocation scheme in this case. Of course, when there are many more virtual registers than physical registers, its performance degrades rapidly as physical register values must be repeatedly spilled and reloaded.

8.3 Local Register Allocation

Local register allocation can involve allocating registers for a single statement or a single basic block.

In (Aho et al, 2007) the authors provide an algorithm that allocates the minimal number of registers required for processing the computation represented by an abstract syntax tree (AST); the algorithm does a post-order traversal of the AST determining the minimum number of registers required and then assigns registers, re-using registers as appropriate, to the computation.[1]

Another strategy is to compute, for each virtual register, a live interval (see section 8.4.1) local to a block. Registers are allocated in the order of the intervals' start positions. When a register must be spilled, we avoid spilling those registers whose values last the longest in the block.

Yet another strategy mixes local and limited global information to good effect. It computes and applies local live intervals to the allocation of registers, but applied to those blocks within the deepest nested loops first – the payoff comes from the fact that instructions more deeply nested within loops are executed much more often.

A more effective register allocation regime considers the entire control flow graph (cfg) for a method's computation, i.e., global register allocation.

8.4 Global Register Allocation

8.4.1 Computing Live Intervals

8.4.1.1 Live Intervals

Global register allocation works with the entire control flow graph for a method in attempting to effectively map virtual registers to physical registers. One wants to minimize spills to memory; and where spills are necessary, one wants to avoid using them within deeply nested loops. The basic tool in global register allocation is the *live interval*, the sequence of instructions for which a virtual register holds a meaningful value.

[1] The algorithm is similar to that used by **`CLEmitter`** for computing the minimum number of locations to allocate to a JVM stack frame for computing expressions in the method.

Live intervals are required by both of the global register algorithms that we consider: linear scan register allocation and register allocation by graph coloring.

In its roughest form, a live interval for a virtual register extends from the first instruction that assigns it a value to the last instruction that uses its value. A more accurate live interval has "holes" in this sequence, where a virtual register does not contain a useful value; for example, a hole occurs from where the previously assigned value was last used (or read) to the next assignment (or write) of a new value.

For example, re-consider the LIR for Factorial's **`computeIter()`**:

```
B0

B1
0: LDC [1] [V32|I]
5: MOVE $a0 [V33|I]
10: MOVE [V32|I] [V34|I]

B2
15: LDC [0] [V35|I]
20: BRANCH [LE] [V33|I] [V35|I] B4

B3
25: LDC [-1] [V36|I]
30: ADD [V33|I] [V36|I] [V37|I]
35: MUL [V34|I] [V33|I] [V38|I]
40: MOVE [V38|I] [V34|I]
45: MOVE [V37|I] [V33|I]
50: BRANCH B2

B4
55: MOVE [V34|I] $v0
60: RETURN $v0
```

The live intervals for this code are illustrated in Figure 8.1. The numbers on the horizontal axis represent instruction ids; recall, instruction ids are assigned at increments of 5 to facilitate the insertion of spill code. The vertical axis is labeled with register ids.

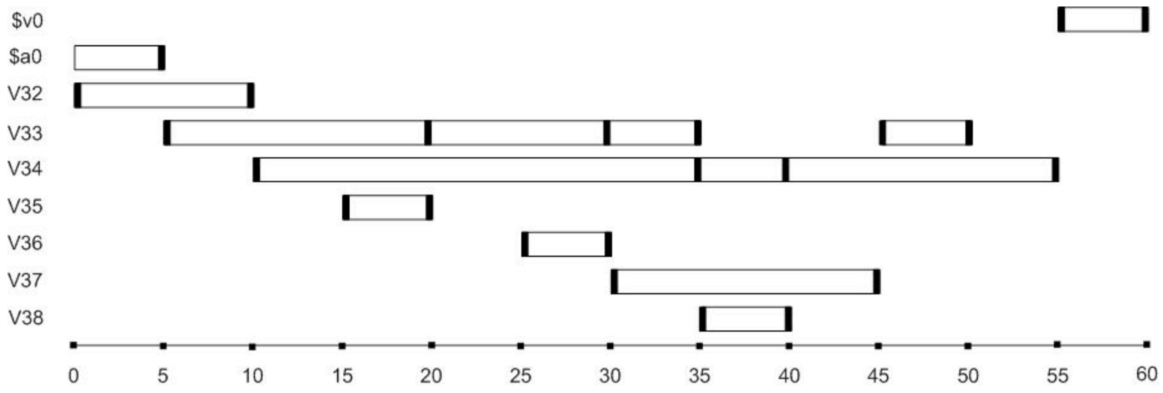

Figure 8.1 Intervals for Factorial.computeIter()

There are two physical registers here: $v0 and $a0[2]; $a0 is the first argument to **`computerIter()`** and the return value is in $v0. There are seven virtual registers: V32 – V38. Recall, we begin numbering the virtual registers at 32 because 0 – 31 are reserved for the 32 general-purpose (physical) registers.

An interval for a virtual register extends *from* where it is defined (it could be defined as an argument; e.g. $a0) to the last time it is used. For example, V37 is defined at instruction position 30 and is last used at instruction position 45.

Some intervals may have *holes*. A *hole* might extend from the instruction following the last instruction using the register up until the next time it is defined. For example, consider virtual register V33. It is defined at position 5 and used at positions 20, 30 and 35. But it is not used again until after it is re-defined at position 45 (notice the loop again). There is a hole at position 40.

Loops can complicate determining an interval. For example, V34 is first defined at position 10, used at 35 and then redefined at 40. But there is a branch (at position 50) back to basic block B2. At some point in the iteration, the conditional branch at position 20 will bring us to block B4, and V34 will be used a last time at position 55. So we say that the interval for V34 extends from position 10 to position 55 inclusively; its value is always meaningful over that interval.

Finally, the darker vertical segments in the intervals identify *use positions*: a position in the interval where ***either*** the register is defined (that is, written) or the register is being used (that is, read). For example, $v0 in Figure 8.1 is defined at position 55 and used at position 60, so there are two segments. But $a0 is never defined in the interval (actually, we assume it was defined in the calling code) but is used at position 5. From the same figure we can see that V34 is either defined or used at positions 10, 35, 40 and 55.

The compiler prints out these intervals using the following format.

```
v0: [55, 60]
a0: [0, 5]
V32: [0, 10]
V33: [5, 35]  [45, 50]
V34: [10, 55]
V35: [15, 20]
V36: [25, 30]
V37: [30, 45]
V38: [35, 40]
```

[2] The compiler's output leaves out the '$' prefix here but we use it to distinguish physical registers.

Here, just the intervals (and not the use positions) are displayed for each register. The notation

[<from> <to>**]**

indicates that the register holds a meaningful value from position <from> to position <to>. For example, the line

```
  V37: [30, 45]
```

says the liveness interval for register V37 ranges from position 30 to position 45. Notice that register V33,

```
  V33: [5, 35]  [45, 50]
```

has an interval with two ranges: from position 5 to position 35, and from position 45 to position 50; there is a hole in-between.

8.4.1.2 Computing Local Live Sets

As a first step in building the live intervals for LIR operands, we compute the set of operands that are live at the start of each block and the set of operands that are (locally) live at the end of each block.

We call the set of operands that are live at the start of a block *liveUse*. LiveUse operands are those operands that are read (or used) before they are written (defined) in the block's instruction sequence; presumably they were defined in a predecessor block (or are arguments to the method).

We call the set of operands that are live at the end of a block *liveDef*. LiveDef operands are those operands that are written to (defined) by some instruction in the block.

Algorithm 8.1 computes liveUse and liveDef for each basic block in a method.

Algorithm 8.1: Computing Local Liveness Information
Input: The control flow graph (cfg) for a method.
Output: Two sets for each basic block: liveUse, registers used before they are overwritten (defined) in the block and liveDef, registers that are defined in the block.

```
 for each block b in cfg.blocks
{
        b.liveUse = {}
        b.liveDef = {}
        for each instruction i in b.instructions
        {
                for each virtual register v in i.readOperands
                {
                        if v ∉ b.liveDef
                        {
                                b.liveUse = b.liveUse ∪ { v }
                        }
                }
                for each virtual register v in i.writeOperands
                {
                        b.liveDef = b.liveDef ∪ { v }
                }
        }
}
```

Our next step is to use this local liveness information to compute global liveness information; for each basic block we want to know which registers are live, both coming in to the block and going out of the block.

8.4.1.3 Computing Global Live Sets

We can compute the set of operands that are live at the beginning and at the end of a block, using a backward dataflow analysis (Aho et al, 2007). We call the set of operands that are live at the start of a block *liveIn*. We call the set of operands that are live at the end of a block *liveOut*.

Notice we iterate through the blocks, and the instruction sequence within each block, in a *backwards* direction.

Algorithm 8.2: Computing Global Liveness Information
Input: The liveDef and liveUse sets for each basic block in a method.
Output: Two sets for each basic block: liveIn, registers that are live coming into the block and liveOut, registers that are live going out of the block.

```
do {
        for each block b in cfg.blocks in reverse order
        {
                b.liveOut = {}
                for each successor s of b
                {
                        b.liveOut = b.liveOut ∪ s.liveIn
                }
        b.liveIn = (b.liveOut – b.liveDef) ∪ b.liveUse
    }
} while any liveOut set was changed
```

We represent the live sets as **`Bitmap`**s from the Java API, indexed by the register number of the operands. Recall, registers numbered 0 to 31 are physical registers and registers 32 and higher are virtual registers. By the time we compute live intervals, we know how many virtual registers we will need for a method's computation. Later, splitting of intervals may create additional virtual registers but we do not need liveness sets for those.

We cannot compute the live sets for a loop in a single pass; the first time we process a loop end block, we have not yet processed the corresponding loop header block – the loop header's block's liveIn set (and so its predecessor blocks' liveOut sets) is computed correctly only in the second pass. This means our algorithm must make d + 1 passes over the blocks where d is the depth of the deepest loop.

We can now use this global live in and live out information to compute accurate live intervals.

8.4.1.4 Building the Intervals

To build the intervals, we make a single pass over the blocks and instructions, again in reverse order. Algorithm 8.3 computes these intervals with both the ranges and the use positions.

Algorithm 8.3: Building Live Intervals
Input: The LIR for a method and the liveIn and liveOut sets for each basic block.
Output: A live interval for each register, with ranges and use positions.

```
for each block b in cfg.blocks in reverse order
{
    int blockFrom = b.firstInstruction.id
    int blockTo = b.lastInstruction.id

    for each register r in b.liveOut
    {
        intervals[r].addOrExtendRange(blockFrom, blockTo)
    }
    for each instruction i in b.instructions in reverse order
    {
        if (i.isAMethodCall)
        {
            for each physical register r
            {
                intervals[r].addRange(i.id, i.id)
            }
        }
        for each virtual register r in i.writeOperands
        {
            intervals[r].firstRange.from = i.id
            intervals[r].addUsePos(i.id)
        }
        for each virtual register r in i.readOperands
        {
            intervals[r]. addOrExtendRange (blockFrom, i.id)
            intervals[r].addUsePos(i.id)
        }
    }
}
```

Before we even look at the LIR instructions, we add ranges for all registers that are live out. These ranges must extend to the end of the block because they are live out. Initially, we define the ranges to extend from the start of the block because they may have been defined in a predecessor; if we later find they are defined (or re-defined) in the block then we shorten this range by overwriting the from position with the defining position.

As we iterate through the instructions of each block, in reverse order, we add or modify ranges:

- When we encounter a subroutine call, we add ranges of length one at the call's position to the intervals of all physical registers. The reason for this is that we must assume the subroutine itself will use these registers and so we would want to force spills.

  In our current implementation, we do not do this step. Rather, we treat all registers as callee-saved registers, making it the responsibility of the called subroutine to save any physical registers that it uses. We leave alternative implementations as exercises.

- If the instruction has a register that is written to, then we adjust the first (most recent) range's start position to be the position of the (writing) instruction, and we record the use position.

- For each register that is read (or used) in the instruction, we add a new range extending to this instruction's position. Initially, the new range begins at the start of the block; a write may cause the start position to be re-adjusted.

  Notice the addOrExtendRange() operation merges contiguous ranges into one.

The careful reader will notice that this algorithm assumes that no virtual register is defined and then not used; this means we don't have local variables that are given values that are never used. We leave dealing with the more peculiar cases as exercises.

Example

Consider how Algorithm 8.3 would deal with basic block B3 of the LIR for **`Factorial.computeIter()`**.

```
B3
25: LDC [-1] [V36|I]
30: ADD [V33|I] [V36|I] [V37|I]
35: MUL [V34|I] [V33|I] [V38|I]
40: MOVE [V38|I] [V34|I]
45: MOVE [V37|I] [V33|I]
50: BRANCH B2
```

The progress of building intervals for basic block B3, as we sequence backwards through its instructions is illustrated in Figure 8.2. (LiveOut contains V33 and V34.)

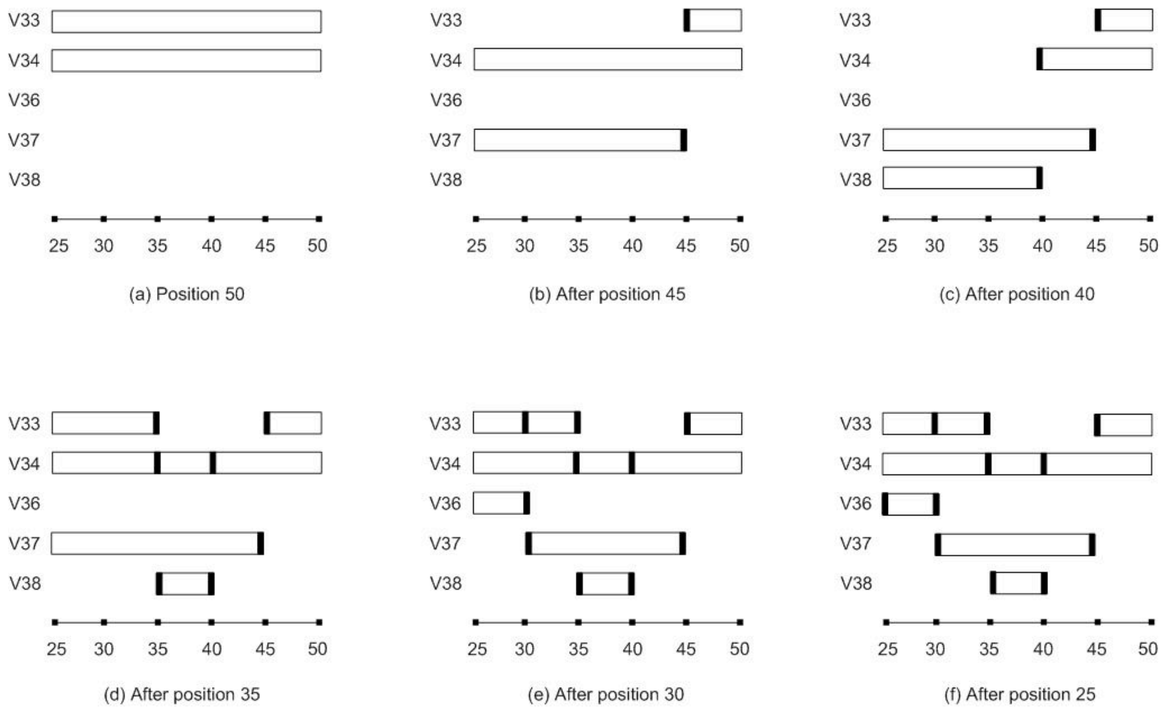

Figure 8.2 Building Intervals for Basic Block B3

At the start of processing B3, virtual registers V33 and V34 are in liveOut and so have intervals with ranges extending from (for now) the start of the block (position 25) to the end of the block (position 50), as illustrated in Figure 8.2 (a). We then sequence through the instructions in reverse order.

- At position 50 the branch instruction has no register operands so no intervals are affected.

- At position 45 the move defines V33 so the range for V33 is shortened to start at 45; V37 is used so we add a range to 37 extending from (for now) the start of the block to position 45. This is illustrated in Figure 8.2 (b).

- At position 40 the move defines V34 so its range is shortened to start at 40; V38 is used so we add a range to V38 extending from (for now) the start of the block to position 40. This is illustrated in Figure 8.2 (c).

- At position 35 the multiply operation defines V38 so its range is shortened to start at 35; V34 is used so we add a range extending from the start of the block to 35, and since it is adjacent to the next segment, we merge the two; V33 is used so we add a segment for V33 from the start of the block to 35. This is illustrated in Figure 8.2 (d).

- At position 30 the add operation defines V37 so we shorten its range to start at 30; V33 is used so we add a use to its interval at 30; V36 is used so we add a range from the start of the block to 30. This is illustrated in Figure 8.2 (e).

- At position 25 the load constant operation defines V36 so we define a use for V36 (definitions are considered uses) at position 25. This is illustrated in Figure 8.2 (f).

Once we have calculated the live intervals for a method, we can set about using them to allocate registers.

8.4.2 Linear Scan Register Allocation

8.4.2.1 Introduction to Linear Scan

Linear scan (Poletto and Sarkar, 1999) is the first register allocation technique we look at and it is the one we have implemented in the JVM code to SPIM translator. Its principal advantage is that it is fast – it really just does a linear scan through the live intervals, in order of their starting positions (earliest first), and maps them to physical registers.

We follow a strategy laid out by Christopher Wimmer in his Master's thesis (Wimmer, 2004), which he used to implement a register allocator for the Oracle HotSpot client JIT compiler. It is fast and it is also intuitive.

8.4.2.2 The Linear Scan Allocation Algorithm

Algorithm 8.4 describes a linear scan register allocation algorithm based on that in (Wimmer, 2004).

First we sort the intervals into increasing order, based on their start positions.

At any time, the algorithm is working with one interval called the *current* interval; this interval has a starting *position*, the **`from`** field of its first range. This position categorizes the remaining intervals into four lists:

1. A list of *unhandled* intervals sorted on their start positions in increasing order. The unhandled list contains intervals starting after position.
2. A list if *active* intervals, whose intervals cover position and so have a physical register assigned to them.
3. A list of *inactive* intervals, each of which start before position and end after position but do not cover position, because position lies in a lifetime hole.
4. A list of *handled* intervals, each of which ends before position or was spilled to memory. These intervals are no longer needed so we do not keep track of them.

Algorithm 8.4: Linear Scan Register Allocation
Input: The cfg for a method with its associated live intervals.
Output: A version of the LIR where all virtual registers are mapped to physical registers. Code for any necessary spills has been inserted into the LIR.

```
unhandled = list of intervals sorted by increasing starting position
active = {}
inactive = {}

while (unhandled ≠ {})
{
    current = first interval removed from unhandled
    position = current.firstRange.from

    for each interval i in active
    {
        if (i.lastRange.to < position) {
            remove it from active // it's handled
        }
        else if (!i.covers(position)) {
            move it from active to inactive
        }
    }
    for each interval i in inactive
    {
```

```
                if (i.lastRange.to < position) {
                        remove it from inactive // it's handled
                }
                else if (i.covers(position)) {
                        move it from inactive to active
                }
        }

        // find a physical register for current
        if (! foundFreeRegisterFor(current)) {
                allocateBlockedRegisterFor(current)
        }

        add current to active
}
```

The algorithm simply scans through the unhandled intervals one at a time, finding a physical register for each; in the first instance it looks for a free physical register and, if it fails at that, it looks for a register to spill. In either case, a split may occur, creating an additional interval and sorting it back into unhandled. Before allocating a register, the algorithm does a little bookkeeping: based on the new position (the start of the current interval), it moves intervals among the lists.

The method, foundFreeRegisterFor() takes an interval as its argument and attempts to map it to a freely available register; if it is successful it returns true; if unsuccessful in finding a free register, it returns false. In the latter case, the linear scan algorithm calls upon the method, allocateBlockedRegisterFor(), which determines which register should be spilled so that it can be allocated to the interval.

Algorithm 8.5 describes the behavior of the method, foundFreeRegisterFor().

Algorithm 8.5: Attempting to allocate a free register
Input: The current interval (from Algorithm 8.4).
Output: true if a free register was assigned to the current interval; false if no free register was found.

```
foundFreeRegisterFor(current)
{
        for each physical register r {
                freePos[r] = maxInt // default
        }
        for each interval in active {
                freePos[i.reg] = 0  // unavailable
        }
        for each interval i in inactive {
                if (i intersects with current) {
                        freePos[i.reg] = min( freePos[i.reg],
                                             next intersection of i with current)
                }
        }
        reg = register with highest freePos
        if (freePos[reg] = 0)
                return false // failure
        else if freePos[reg] > current.lastRange.to then {
                // reg available for whole of current
                current.r = reg // assigned to current interval
        }
        else {
                // register available for first part of current
                current.r = reg // assigned to current interval
                // split current at optimal position before freePos[reg]
                put current with the split off part back on unhandled
        }
        return true // success
}
```

The array freePos[] is used to compute the next position that each register is used; the register is free up until that position. The element freePos[r] records this position for the physical register (numbered) r.

By default, all physical registers are free for the lifetime of the method; hence the freepost is maxInt. But none of the physical registers assigned to intervals that are on the active list are free; hence the freepost is 0. But for registers assigned to intervals that are inactive (and so in a hole) are assigned a freePos equal to the next position that the current interval intersects with the interval in question, that is the position where both intervals would need the same register; the register is free to be assigned to the current interval up until that position. Because the same physical register may (because of previous splits) have several next use positions, we choose the closest (the minimum) for freePos.

So, when we chose the register with the highest freePos, we are choosing that register that will be free the longest.

If the chosen register has a freePos of 0 then neither it nor any other register is free and so we return false to signal we will have to spill some register. But if its freePos is greater than the to position of current's last range then it is available for the duration of current and so is assigned to current. Otherwise the interval we are considering is in a hole; indeed it is in a hole that extends the furthest of any other interval in a hole. Its register is available to the current interval up until that position. So we *split* the current interval at *some optimal position* between the current position and that freePos, we assign the interval's register (temporarily) to the first split off part of current, and we put current with the remaining split off part back on unhandled for further processing. So we make the best use of registers that are assigned to intervals that are currently in holes.

If current and the candidate interval don't intersect at all then both intervals may make use of the same physical register with no conflict; one interval is in a hole while the other is making use of the shared register.

When we say we split the current interval at *some optimal position*, we are choosing how much of the hole to take up for current's purposes. One could say we want as much as possible, that is all the way up to freePos. But, in the next section we shall see that this is not always wise.

Now, if no register is free to be assigned to current even for a portion of its lifetime, then we must spill some interval to a memory location on the stack frame. We use Algorithm 8.6 (Wimmer, 2004) to select an interval for spilling and to assign the freed register to current. The algorithm chooses to spill that register that is not used for the longest time – that register next used at the highest position in the code.

Algorithm 8.6: Spill and allocate a blocked register
Input: The current interval (from Algorithm 8.4).
Output: A register is spilled and assigned to the current interval.

```
allocateBlockedRegisterFor(current)
{
        for each physical register r {
                usePos[r] = blockPos[r] = maxInt
        }
        for each non-fixed interval i in active {
                usePos[i.reg] = min( usePos[i.reg],
                                     next usage of i after current.firstRange.from )
        }
        for each non-fixed interval i in inactive {
                if (i intersects with current) {
                        usePos[i.reg] = min( usePos[i.reg],
                                             next usage of i after current.firstRange.from )
                }
        }
        for each fixed interval i in active {
                usePos[i.reg] = blockPos[i.reg] = 0
        }
        for each fixed interval i in inactive {
```

```
        blockPos[i.reg] = next intersection if i with current
        usePos[i.reg] = min( usePos[i.reg], blockPos[i.reg] )
    }

    reg = register with highest usePos
    if (usePos[reg] < first usage of current) {
        // all intervals are used before current => spill current itself
        assign a spill slot on stack frame to current
        split current at an optimal position before first use position requiring
        a register
    } else if (blockPos[reg] > current.lastRange.to) {
        // spilling frees reg for all of current
        assign a spill slot on stack frame to child interval for reg
        assign register reg to interval current
        split, assign stack slot, and spill intersecting active and
            inactive intervals for reg
    } else {
        // spilling frees reg for first part of current
        assign a spill slot on stack frame to child interval for reg
        assign reg to interval current
        split current at optimal position before blockPos[reg]
        split, assign stack slot, and spill intersecting active and
            inactive intervals for reg

    }
}
```

This algorithm collects two positions for each physical register r by iterating through all active and inactive intervals:

1. usePos[r] records the position where an interval with register r assigned is used next. If more than one use position is available, the minimum is used. It is used in selecting a spill candidate.
2. blockPos[r] records a (minimum) hard limit where register r cannot be freed by spilling. Since this is a candidate value for usePos, usePos can never be higher than this hard blockPos.

The register with the highest usePos is selected as a candidate for spilling. Based on the use positions, the algorithm identifies three possibilities:

1. It is better to spill current if its first use position is found after the candidate with the highest usePos. We split current before its first use position where it must be reloaded; the active and inactive intervals are left untouched. This case also applies if all registers are blocked at a call site.
2. Otherwise, current is assigned the selected register. All inactive and active intervals for this register intersecting with current are split before the start of

current and spilled. We needn't consider these split children further because (of the case condition) they don't have a register assigned. But if they have use positions requiring a register, they will have to be reloaded to some register so they are split a second time before the use positions and the second split children are sorted back into unhandled. They will get a register assigned when the allocator has advanced to their position.

3. If the selected register has a blockPos in the middle of current, then it is not available for current's entire lifetime; current is split before blockPos and the sorted child is sorted into unhandled (That the current interval starts before a use is probably the result of a previous split.)

Either current gets spilled or current is assigned a register. Many new split children may be sorted into unhandled, but the allocator always advances position and so will eventually terminate.

8.4.2.3Splitting Intervals

In some cases, where an interval cannot be assigned a physical register for its entire lifetime, we will want to split intervals.

Consider the interval for virtual register V33:

```
  V33: [5, 35]  [45, 50]
```

Originally, this interval is modeled by the NInterval object in Figure 8.3 (a).

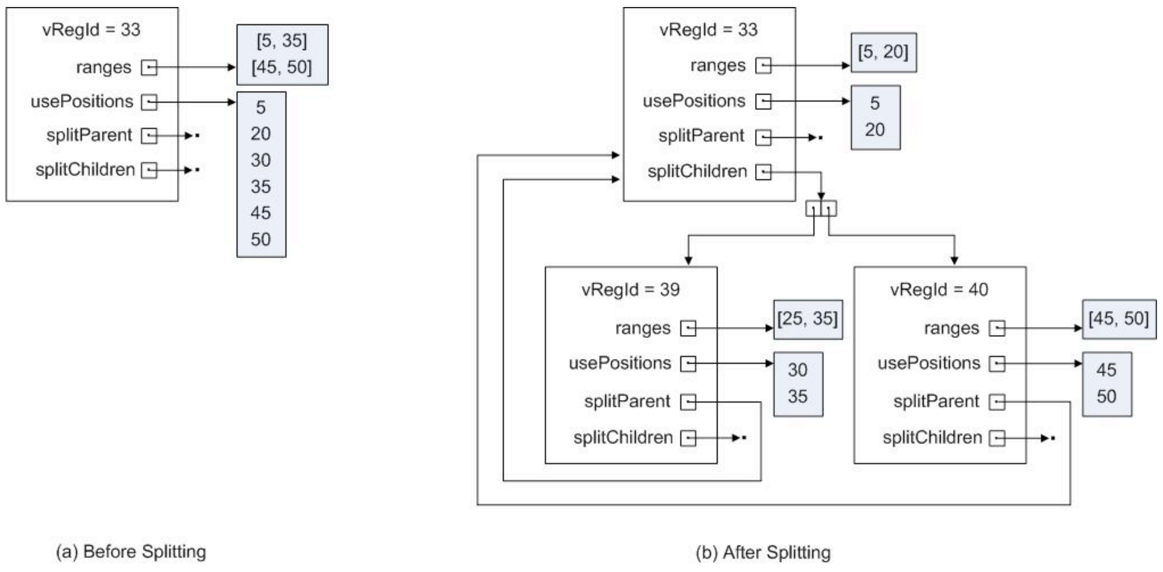

Figure 8.3 The Splitting of Interval 33

Then, we split the interval twice:

1. Firstly, the interval is split at position 20. This creates a new interval 39 (the next available virtual register number) with all ranges and use positions above 20. The original range [5, 35] is split into the range [5, 20] for interval 33 and the range [25, 35] for interval 39.
2. Next, the interval 39 is split at position 40, creating the new interval 40. Although the split occurred at position 40, interval 39 now ends at position 35 and interval 40 starts at position 45 since neither interval is live between 35 and 45.

The result of the two splits is illustrated in Figure 8.3 (b). Each interval maintains an array of split children.

8.4.2.4 Choosing an Optimal Split Position

Once we have burned through all of our registers we will be forced to split intervals. As we have seen above, we may split current to temporarily assign it a physical register that is attached to an inactive interval. We may split and spill intervals that are active. Usually, we are offered a range of positions where we may split an interval; the position where an interval is spilled or reloaded can be moved to a lower position. Where exactly we choose to split an interval can affect the quality of the code.

Wimmer (Wimmer, 2004) proposes the following three heuristics to apply when choosing a split position:

1. Move the split position out of loops. Code in loops are executed much more often than code outside of loops, so it is best to move register spills and register reloads outside of these loops.
2. Move the split position to block boundaries. Data flow resolution (in the next section) takes care of spills and reloads at block boundaries automatically .
3. Move split positions to positions that are not multiples of five. These positions are not occupied by normal LIR operations and so are available for spills and reloads.

These rules don't guarantee optimal code but they go along way to improving the code quality.

8.4.2.5 Resolving the Data Flow, Again

The splitting of intervals that occurs as a result of register allocation may cause inconsistencies in the resulting code.

For example, consider the follow of data in an in-then-else statement,

```
if ( <condition> ) {
      <then part>
} else {
      <else part>
}
<subsequent code>
```

Consider the case where the register allocator must spill a register (say, for example, $s5) in allocating registers to the LIR code for the <else part>; somewhere in the LIR code for the <else part> is a move instruction (e.g. a store) to spill the value of $s5 to a slot on the run-time stack. If the <subsequent code> makes use of the value in $s5 then it will know that the value has been spilled to the stack because the interval will extend at least to this next use; it will use a move to load that value back into a (perhaps different) register.

But what if the flow of control passes through the <then part> code? Again, <subsequent code> will expect the value of $s5 to have been spilled to the stack slot; it will load a value that has not been stored there.

For this reason, an extra store of $s5 must be inserted at the end of the <then part> code so that the (spilled) state of $s5 is consistent in the subsequent code.

Algorithm 8.7 (Wimmer, 2004) and (Traub et al, 1998) inserts these necessary extra moves.

Algorithm 8.7: Resolve Data Flow
Input: The version of the cfg after register allocation (Algorithm 8.4).
Output: The same LIR but with moves inserted to resolve changes in the data flow.

```
resolveDataFlow() {
    MoveResolver resolver; // used for ordering and inserting moves into the LIR\

    for each block b in cfg.blocks {
        for each successor s of b {
            // collect all necessary resolving moves between b and s
            for each operand r in s.liveIn {
                parentInterval = cfg.intervals[opr]
                fromInterval = parentInterval.childAt(b.lastOP.id)
                toInterval = parentInterval.childAt(s.firstOP.id)
                if (fromInterval ≠ toInterval) {
                    resolver.addMapping( fromInterval, toInterval )
                }
            }

            // The moves are inserted either at the end of block b
            // or the start of block s, depending on the control flow.
            resolver.findInsertPosition(b,s)
```

```
                    // insert all moves in an order that insures registers with values
                    // used later are not overwritten.
                    resolver.resolveMappings()
                }
        }
}
```

The algorithm iterates through the blocks, looking at their successors that define the block boundaries. It identifies the original virtual registers whose intervals spanned those boundaries; this is true for intervals for virtual registers that are live-in across the boundaries. Finally, it identifies intervals across these boundaries that were split during register allocation by comparing the child intervals on either side: if the child intervals differ then a split has occurred and so we want to insert resolving moves.

To insert the resolving moves,

- The method findInsertPosition() determines whether to put the moves at the end of the from block (b) – of course before any branches – or at the start of the to block (s), depending on the control flow.
- The method resolveMappings() inserts any moves that are present, making sure to order them so that one does not overwrite a register before its value has been spilled.

Notice the similarity to the algorithm for Phi-resolution when translating HIR to LIR in Chapter 7.

8.4.2.6 Physical Register Assignment in the LIR

The final strep in Wimmer's strategy replaces the virtual registers in the LIR with assigned physical registers. Algorithm 8.8 (Wimmer, 2004) does this register assignment.

Algorithm 8.8: Assign Physical Register Numbers
Input: The version of the cfg after resolution (Algorithm 8.7).
Output: The same LIR but with virtual registers replaced by physical registers.

```
for each block b in cfg.blocks {
        for each instruction i in b.instructions {
                for each virtual register r in i.operands
                        // determine new operand
                        physicalRegister = intervals[r].childAt(i.id).assignedReg
                        replace r with physicalRegister in i
                }

                if ( i is a move with the target equal to the source) {
                        remove i from b.instructions
                }
        }
}
```

Now the LIR refers only to physical registers and is ready for translation to SPIM.

8.4.2.7 Some Improvements to the LIR

An examination of the LIR after register allocation (using any global allocation strategy) makes it clear that many (often unnecessary) moves are generated.

Because we have not made fixed registers, e.g. $a0 - $a3 and $v0 (and $v1), available for allocation, there is much movement among these and virtual registers in the original LIR; this leads to many meaningless moves from one register to another once physical registers are allocated. Bringing these physical registers (modeled as fixed intervals) into play should eliminate our overreliance on other general-purpose registers and eliminate many moves among them.

There is much room for spill optimization. When a variable is defined once and then used several times, it need be spilled just once even if the interval modeling the virtual register holding the value is split several times. A dirty bit may be used to keep track of definitions versus uses. Certain arguments to methods, e.g. $a0 when it contains the address of an object for **`this`**, need be spilled just once also.

If all of a block's predecessors end with the same sequence of move instruction, they may be moved to the start of the block itself. Likewise, if all of a block's successors start with the same sequence of move instructions, they can be moved to the end of the (predecessor) block. Such moves are most likely to be introduced either by Phi-resolution (Chapter 7) or by data flow resolution (section 8.4.2.4). Wimmer makes the point that this is best done after virtual registers have been replaced by physical registers because this allows us to combine moves involving like physical registers even if they originated from differing virtual registers.[3]

By examining the output of the register allocation process, one will find many places for improvement. It is best to formulate algorithms that are as general as possible in making these improvements.

8.4.2.8 What is in the Code Tree; what is left Undone

Much of Wimmer's strategy, recounted above, is implemented in our code. The following issues are not addressed and so are left as exercises.

[3] Order is often an important consideration when applying any optimizations; subsequent optimizations may take advantage of preceding ones.

- We do not deal with physical registers in the register allocation process. Although we make use of particular fixed registers, for example \$a0 - \$a3 for holding arguments passed to methods and \$v0 for holding return values, we do not otherwise make these available for allocation to other computations. Moreover, when we allocate \$t0 - \$t9 and \$s0 - \$s7 we assume they are callee-saved. The Wimmer algorithms treat them as caller-saved and depend of the (short) intervals at calls to insure they are spilled before the calls, and reloaded after the calls; otherwise they are treated like any other general purpose registers and are available to the allocator.
- Our code does not use the heuristics in section 8.4.2.4 to find the optimal split position.
- Our code does not optimize spills as discussed in section 8.4.2.7.
- Our code does not move sequences of like moves between successors and predecessors, as described in section 8.4.2.7.

8.4.3 Register Allocation by Graph Coloring

8.4.3.1 Introduction to Graph Coloring Register Allocation

In graph coloring register allocation, we start with an *interference graph* built from the liveness intervals. An interference graph consists of a set of nodes, one for each virtual register or liveness interval, and a set of edges. There is an edge between two nodes if the corresponding intervals interfere, that is if they are live at the same time.

For example, reconsider the live intervals for the virtual registers V37 – V38 of **`Factorial.computeIter()`**, originally in Figure 8.1 but repeated here in Figure 8.4

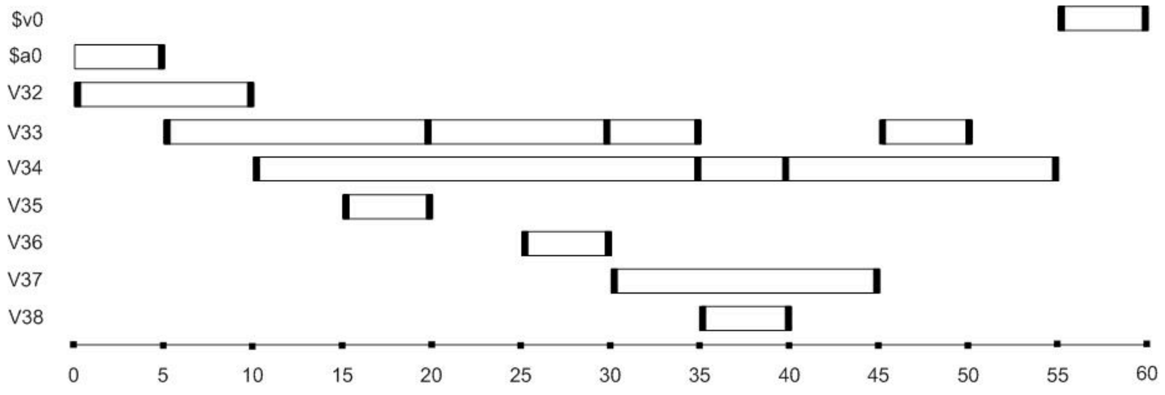

Figure 8.4 Intervals for Factorial.computeIter(), again

Figure 8.5 shows the corresponding interference graph.

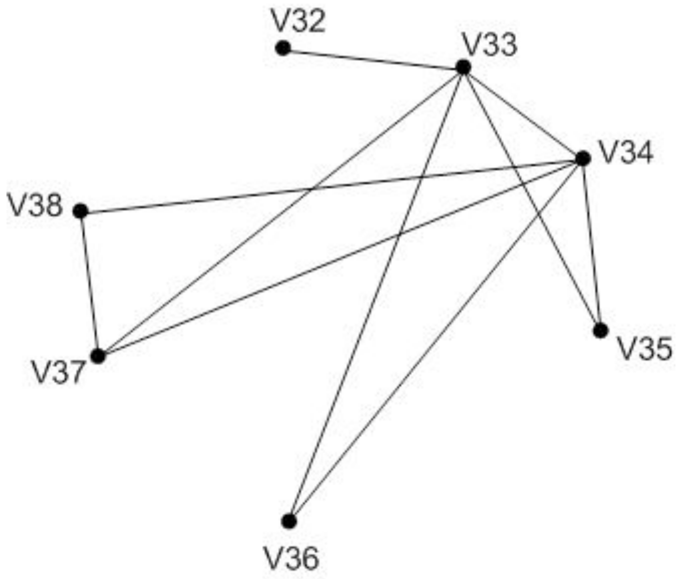

Figure 8.5 Interference Graph for Intervals for Factorial.computeIter()

We need not draw an arc from the node for a virtual register we are using to the node for the virtual register we are defining. For example, at position 10, the instruction

```
10: MOVE [V32|I] [V34|I]
```

uses V32 to define virtual register V43; so the interval for V32 ends where the interval for V34 begins. So we draw no edge from V32 to V34.

Register allocation then becomes coloring the graph but using physical registers as the colors. The question becomes, can the graph be "colored" such that no two adjacent nodes (that is no two intervals that are live at the same time) share the same physical register. John Cocke was the first person to see that register allocation could be modeled as graph coloring (Kennedy, 1971). Gregory Chaitin and his colleagues (Chaitin et al, 1981 and Chaitin, 1982) implemented such an allocator at IBM in 1981.

We say that a graph has an R-coloring if it can be colored using R distinct colors, or in our case R distinct physical registers. To exhaustively find such an R-coloring for $R \geq 2$ has long been known to be NP-complete. But there are two heuristics available to us for simplifying the graph.

1. The first, called *the degree < R rule*, derives from the fact that a graph with a node of degree < R (that is a node with < R adjacent nodes) is R-colorable if and only if the graph with that node removed is R-colorable. We may use this rule to prune the graph, removing one node of degree < R at a time and pushing it onto a stack; removing nodes from the graph removes corresponding edges, potentially

creating more nodes with degree < R. We continue removing nodes until either all nodes have been pruned or until we reach a state where all remaining nodes have degrees ≥ R.

2. The second is called the *optimistic heuristic* and allows us to continue pruning nodes even with degree ≥ R. We use a heuristic function spillCost() to find a the node having the smallest cost of spilling its associated virtual register; we mark that register for possible spilling and remove the node (and its edges) and push it onto the stack in the hope that we will not really have to spill it later. (Just because a node has degree ≥ R does not mean the nodes need different colors.)

Let us attempt to prune the interference graph in Figure 8.5 where R = 3, meaning we have three physical registers to work with. In this case, we need invoke just the first degree < R rule. The steps to pruning the graph are illustrated in Figure 8.6.

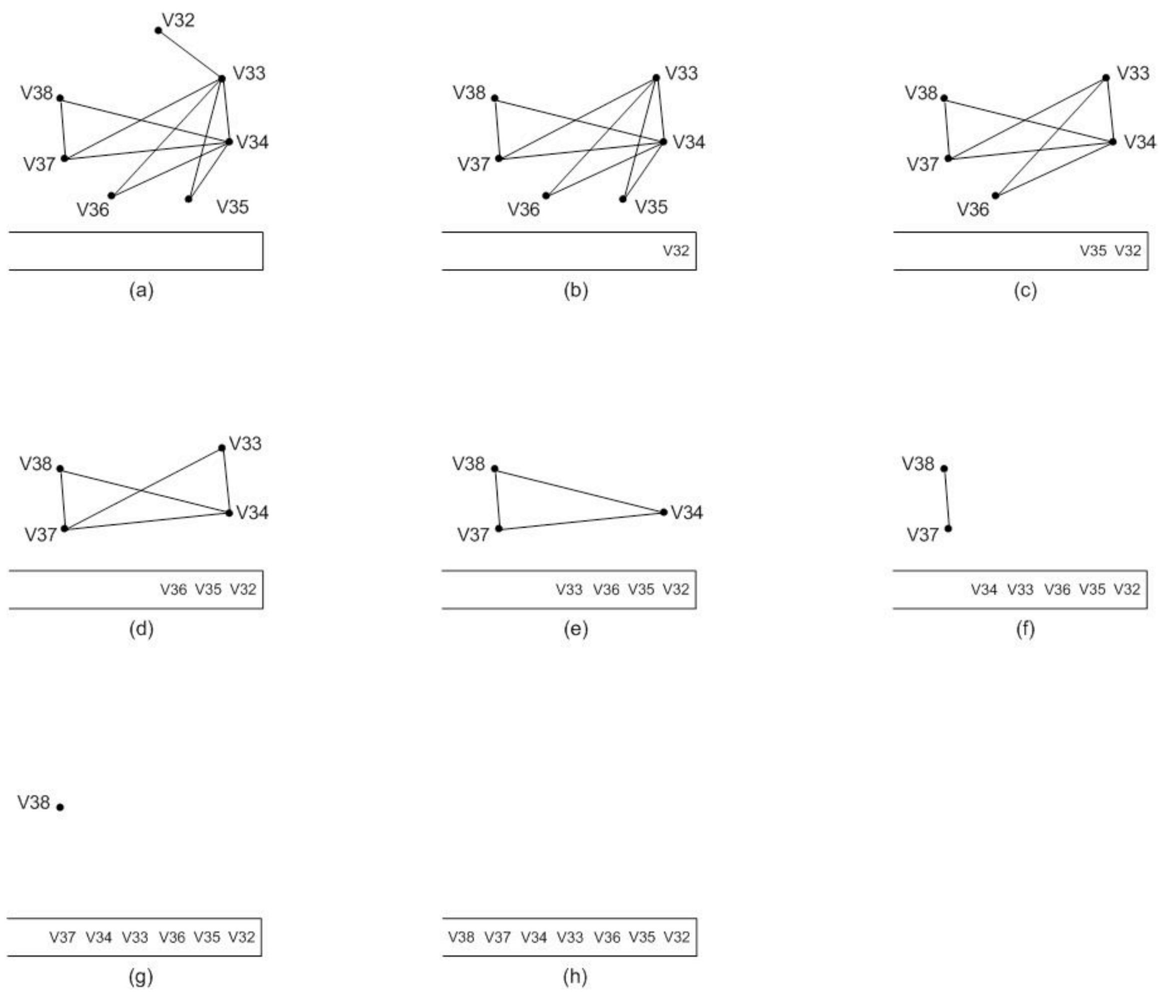

Figure 8.6 Pruning the Interference Graph

The figure pretty much speaks for itself. We start with V32, which (in Figure 8.6 (a)) has just one adjacent interval, remove it and push it onto the stack to give us the graph in Figure 8.6 (b). We continue in this way until all nodes have been removed and pushed onto the stack. Removing nodes with degree < 3 causes edges to be removed and so more nodes end up with degree < 3.

We may then pop the virtual registers off of the list, one at a time, and try to assign physical register numbers (r1, r2, or r3) to each in such a way that adjacent virtual registers (in the graph) are never assigned the same physical register. A possible assignment is

V38	r1
V37	r2
V34	r3

V33	r1
V36	r2
V35	r2
V32	r2

Imposing this mapping onto our LIR for **`Factorial.computeIter()`** gives us:

```
B0

B1
0: LDC [1] r2
5: MOVE $a0 r1
10: MOVE r2 r3

B2
15: LDC [0] r2
20: BRANCH [LE] r1 r2 B4

B3
25: LDC [-1] r2
30: ADD r1 r2 r2
35: MUL r3 r1 r1
40: MOVE r1 r3
45: MOVE r2 r1
50: BRANCH B2

B4
55: MOVE r3 $v0
60: RETURN $v0
```

The algorithm attempts to coalesce virtual registers where there is a move from one to the next and the two registers' intervals do not conflict.

8.4.3.2 The Graph Coloring Register Allocation Algorithm

Algorithm 8.9 performs graph coloring register allocation based on that of (Muchnick, 1997). The algorithm attempts to coalesce virtual registers where there is a move from one to the next and the two registers' intervals do not conflict. Also, once spill code is generated, it repeats the register allocation process. It does this repeatedly until it succeeds in assigning physical registers to virtual registers without introducing additional spills.

Algorithm 8.9: Graph Coloring Register Allocation
Input: The LIR for a method that makes use of virtual registers.
Output: The same LIR but with virtual registers replaced by physical registers.

```
do {
    do {
        buildIntervals()
        buildInterferenceGraph()
    } while ( coalesceRegistersSuccessful() )

    buildAdjacencyLists()

    computeSpillCosts()

    pruneGraph()

    registersAssignedSuccessfully = assignRegistersSuccessful()

    if ( ! registersAssignedSuccessfully )
        generateSpillCode()

} until ( registersAssignedSuccessfully )
```

This algorithm looks like it could go on for a long time, but it usually terminates after at most two or three iterations.

8.4.3.3 Register Coalescing

The algorithm attempts to coalesce virtual registers where there is a move from one to the next and the two registers' intervals do not conflict. Coalescing registers reduces both the number of virtual registers and the number of moves. The method, coalesceRegistersSuccessful() returns true if it is able to coalesce two registers and false otherwise; this Boolean result insures that any register coalescing is followed by a rebuilding of the intervals and the interference graph.

Register coalescing also makes for longer intervals and could render the graph uncolorable. (Briggs, 1994) and (George and Appel, 1996) propose more conservative conditions under which we may safely coalesce registers, without making the graph uncolorable. Briggs proposes that two nodes in the graph may be coalesced if the resulting node will have fewer than R neighbors of *significant degree*, (having R or more edges). George and Appel propose that two nodes may be coalesced if for every neighbor t of one of those nodes, either t already interferes with the other node (so no additional edges are added) or t is of insignificant degree. Appel (Appel, 2002) points out that these rules may prevent the safe removal of certain moves but that extra moves are less costly than spills.

8.4.3.4 Representing the Interference Graph

The representation of the interference graph is driven by the kind of queries one wants to make of it. Our algorithm wants to ask two things:

1. Whether or not one node is adjacent to another. We want to know this when we coalesce registers and when we want to assign registers.
2. How many nodes, and which nodes, are adjacent to a given node.

The first is best answered by a Boolean adjacency matrix: a matrix with a row and column for each register; adj[i,j] is true if nodes i and j are adjacent and false otherwise. A lower-triangular matrix is sufficient because there is an ordering on the nodes. For example, our interference graph in Figure 8.5 might be captured using the following lower-triangular matrix.

	V32	V33	V34	V35	V36	V37
V33	T					
V34	F	T				
V35	F	T	T			
V36	F	T	F	F		
V37	F	T	T	F	F	
V38	F	F	T	F	F	T

Method buildIntervals() might build such an adjacency matrix.

The second is best answered by a couple of vectors, each indexed by the register number: a vector of integers recording the numbers of neighbors, and a vector of lists of neighboring nodes. These vectors are easily computed from the adjacency matrix. Method buildAdjacencyLists() might build these vectors.

8.4.3.5 Determining Spill Cost

During the pruning process we may reach a state where only nodes with degree ≥ R remain in the graph. In this state, our algorithm must choose a node with the smallest spill cost.

The spill cost of a register certainly depends on the loop depths of the positions where the register must be stored to or loaded from memory; (Muchnick, 1997) suggests summing up the uses and definitions, using a factor of 10^{depth} for taking loop depth into account. Muchnick also suggests re-computing the value for a register when it is cheaper than spilling and reloading that same value.

8.4.3.6 Pre-coloring Nodes

Our example in section 8.4.3.1 does not address fixed intervals. (Muchnick, 1997) suggests *pre-coloring* nodes in the interference graph that correspond to these pre-chosen physical registers, e.g. $a0 - $a3, $v1 and $v2, and bring them into the allocation process so that they may be used for other purposes when they are not needed for their specialized functions. Register allocation designers take an aggressive approach to register allocation, where all registers are in play.

8.4.3.7 Comparison to Linear Scan Register Allocation

The complexity of linear scan register allocation is roughly linear with respect to n -- the number of virtual registers, where the complexity of graph coloring register allocation is roughly n^2. At the same time, graph coloring register allocation is claimed to do a much better job at allocating physical registers, producing faster code. For this reason, linear scan is often sold as a solution to many just-in-time compiling strategies where compilation occurs at run time and graph coloring is sold as a strategy where code quality is more important than compile time. For example, in the Oracle HotSpot compiler, a client version uses linear scan and a server version uses graph coloring (Wimmer, 2004).

Even so, there has been a kind of duel between advocates for the two strategies, where linear scan register allocation developers are improving the quality of register assignment, e.g. (Traub, 1998) and graph coloring register allocation developers are improving the speed of register allocation, e.g. (Cooper and Dasgupta, 2006).

Further Reading

The linear scan algorithm discussed in this chapter relies extensively on (Wimmer, 2004) and (Traub et al, 1998). (Wimmer and Franz, 2010) describe a version of the linear scan that operates on code adhering to SSA.

Graph coloring register allocation was first introduced in (Chaitin, et al, 1981) and (Chaitin, 1982). Two texts that describe graph coloring register allocation in great detail are (Appel, 2002) and (Muchnick, 1997). The performance of graph coloring register allocation is addressed in (Cooper and Dasgupta, 2006).

Exercises

8.1 Implement a local register allocation strategy that works with liveness intervals computed only for local individual basic blocks.

8.2 Implement a local register allocation strategy that works with individual basic blocks but that gives preference to blocks that are most nested within loops.

8.3 Modify the linear scan algorithm described in section 8.3 so that it deals effectively with virtual registers that are defined but are never used.

8.4 Modify our code to deal with physical registers in the register allocation process. Treat all registers as caller-saved registers and bring registers such as $a0 - $a3 for holding arguments passed to methods and $v0 for holding return values into the greater allocation process.

8.5 Modify our code to use the heuristics in section 8.4.2.4 to find the optimal split position.

8.6 Modify our code to optimize spills as discussed in section 8.4.2.7.

8.7 Modify our code to move sequences of like moves between successors and predecessors, as described in section 8.4.2.7.

8.8 Implement the linear scan register allocation algorithm that operates on LIR in SSA form and is described in (Wimmer and Franz, 2010).

8.9 Repeat the example register allocation performed by hand in section 8.4.3.1 but where R = 2; that is, where we have two physical registers to work with.

8.10 Implement Algorithm 8.9 for performing graph coloring register allocation on our LIR.

8.11 Modify the coalesceRegistersSuccessful() method to make use of the more conservative (Briggs, 1994) condition for identifying registers that may be coalesced.

8.12 Modify the coalesceRegistersSuccessful() method to make use of the more conservative (George and Appel, 1996) condition for identifying registers that may be coalesced.

8.13 Compare the run-time speed of your linear scan register allocator and your graph coloring register allocator. How does the code produced compare?

8.14 Try pre-coloring fixed intervals, bringing more physical registers into your graph coloring register allocation program.