Register Allocation

Outline

Introduction

Register allocation is the process of assigning as many local variables and temporaries to physical registers as possible

Introduction

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 memory, the faster our programs will run

Introduction

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 memory, the faster our programs will run

With respect to our LIR, we wish to assign physical registers to each of the virtual registers that serve as operands to instructions

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 memory, the faster our programs will run

With respect to our LIR, we wish to assign physical registers to each of the virtual registers that serve as operands to instructions

But there are often fewer physical registers than there are virtual registers

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 memory, the faster our programs will run

With respect to our LIR, we wish to assign physical registers to each of the virtual registers that serve as operands to instructions

But there are often fewer physical registers than there are virtual registers

Sometimes, 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

Introduction

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 memory, the faster our programs will run

With respect to our LIR, we wish to assign physical registers to each of the virtual registers that serve as operands to instructions

But there are often fewer physical registers than there are virtual registers

Sometimes, 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; we would like to minimize this spilling (and reloading) of values to and from memory

Introduction

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

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

The problem of register allocation is NP-complete in general

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

The problem of register allocation is NP-complete in general

Register allocation that focuses on just a single basic block, or even just a single statement, is said to be local

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

The problem of register allocation is NP-complete in general

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

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

The problem of register allocation is NP-complete in general

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

We will look at a naïve (local) strategy as well as a global strategy based on graph coloring

Naïve Register Allocation

The naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning physical registers to virtual registers

Naïve Register Allocation

The naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning physical 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

Naïve Register Allocation

The naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning physical 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 for 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

Naïve Register Allocation

The naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning physical 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 for 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

Naïve Register Allocation

The naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning physical 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 for 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 strategy 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

Naïve Register Allocation

The naïve register allocation strategy simply sequences through the operations in the (LIR) code, assigning physical 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 for 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 strategy 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

When there are many more virtual registers than physical registers, which is always the case, performance of the naïve strategy degrades rapidly as physical register values must be repeatedly spilled and reloaded

Naïve Register Allocation

Naïve Register Allocation

Following is the SPIM code generated by *j--* using the naïve register allocation strategy (and just three physical registers $t0, $t1, and $t2) for `Factorial.computeIter()` method

```
.text

Factorial.computeIter:
    subu    $sp,$sp,48   # Stack frame is 48 bytes long
    sw      $ra,44($sp)         # Save return address
    sw      $fp,40($sp)         # Save frame pointer
    sw      $t0,36($sp)         # Save register $t0
    sw      $t1,32($sp)         # Save register $t1
    sw      $t2,28($sp)         # Save register $t2
    addiu   $fp,$sp,44   # Save frame pointer

Factorial.computeIter.0:

Factorial.computeIter.1:
    li $t0,1
    sw $t0,0($sp)
    move $t1,$a0
    sw $t1,8($sp)
    lw $t0,0($sp)
    move $t2,$t0
    sw $t2,16($sp)

Factorial.computeIter.2:
    li $t0,0
    sw $t0,4($sp)
    lw $t1,8($sp)
    lw $t0,4($sp)
    ble $t1,$t0,Factorial.computeIter.4
    j Factorial.computeIter.3
```

Naïve Register Allocation

Naïve Register Allocation

```
Factorial.computeIter.3:
    li $t1,-1
    sw $t1,12($sp)
    lw $t1,8($sp)
    lw $t1,12($sp)
    add $t2,$t1,$t1
    sw $t2,20($sp)
    lw $t2,16($sp)
    lw $t1,8($sp)
    mul $t0,$t2,$t1
    sw $t0,24($sp)
    lw $t0,24($sp)
    move $t2,$t0
    sw $t2,16($sp)
    lw $t2,20($sp)
    move $t1,$t2
    sw $t1,8($sp)
    j Factorial.computeIter.2

Factorial.computeIter.4:
    lw $t2,16($sp)
    move $v0,$t2
    j Factorial.computeIter.restore

Factorial.computeIter.restore:
    lw      $ra,44($sp)          # Restore return address
    lw      $fp,40($sp)          # Restore frame pointer
    lw      $t0,36($sp)          # Restore register $t0
    lw      $t1,32($sp)          # Restore register $t1
    lw      $t2,28($sp)          # Restore register $t2
    addiu   $sp,$sp,48   # Pop stack
    jr      $ra          # Return to caller
```

Register Allocation by Graph Coloring

Global register allocation works with a method's entire control-flow graph to map virtual registers to physical registers

Register Allocation by Graph Coloring

Global register allocation works with a method's entire control-flow graph to map virtual registers to physical registers

One wants to minimize spills to memory; where spills are necessary, one wants to avoid using them within deeply nested loops

Register Allocation by Graph Coloring

Global register allocation works with a method's entire control-flow graph to map virtual registers to physical registers

One wants to minimize spills to memory; where spills are necessary, one wants to avoid using them within deeply nested loops

The basic tool in global register allocation is the liveness interval, the sequence of instructions for which a virtual register holds a meaningful value

Register Allocation by Graph Coloring

Global register allocation works with a method's entire control-flow graph to map virtual registers to physical registers

One wants to minimize spills to memory; where spills are necessary, one wants to avoid using them within deeply nested loops

The basic tool in global register allocation is the liveness interval, the sequence of instructions for which a virtual register holds a meaningful value
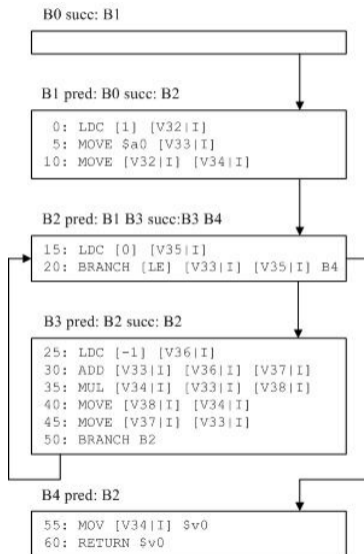
A liveness interval for a virtual register extends from the first instruction that assigns it a value to the last instruction that uses its value

Register Allocation by Graph Coloring

Global register allocation works with a method's entire control-flow graph to map virtual registers to physical registers

One wants to minimize spills to memory; where spills are necessary, one wants to avoid using them within deeply nested loops

The basic tool in global register allocation is the liveness interval, the sequence of instructions for which a virtual register holds a meaningful value

A liveness 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 liveness interval has "holes" in the 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

## Register Allocation by Graph Coloring

Consider the control-flow graph for `Factorial.computeIter()`

B0 succ: B1

```
```

B1 pred: B0 succ: B2

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

B2 pred: B1 B3 succ:B3 B4

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

B3 pred: B2 succ: B2

```
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 pred: B2

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

Register Allocation by Graph Coloring

The liveness intervals for the above code are illustrated in the following figure
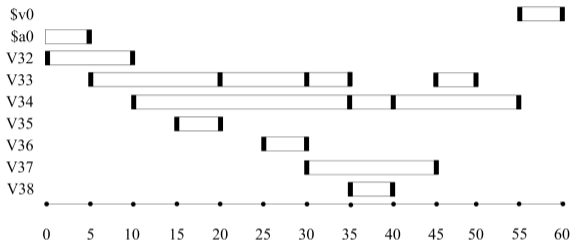
The numbers on the horizontal axis represent instruction ids and the vertical axis is labeled with register IDs

The darker vertical segments in the intervals identify use positions: a use position is a position in the interval where either the register is defined (ie, written) or the register is being used (ie, read)

Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

We first compute, for each block, two local liveness sets: liveUse and liveDef

Register Allocation by Graph Coloring

We first compute, for each block, two local liveness sets: liveUse and liveDef

LiveUse operands are those operands that are read (or used) before they are written (defined) in the block's instruction sequence

Register Allocation by Graph Coloring

We first compute, for each block, two local liveness sets: liveUse and liveDef

LiveUse operands are those operands that are read (or used) before they are written (defined) in the block's instruction sequence

LiveDef operands are those operands that are written to (defined) by some instruction in the block

Register Allocation by Graph Coloring

We first compute, for each block, two local liveness sets: liveUse and liveDef

LiveUse operands are those operands that are read (or used) before they are written (defined) in the block's instruction sequence

LiveDef operands are those operands that are written to (defined) by some instruction in the block

---

Algorithm Computing Local Liveness Information

Input: The control-flow graph g 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 block b in g.blocks **do**
    Set b.liveUse ← {}
    Set b.liveDef ← {}
    for instruction i in b.instructions **do**
      for virtual register v in i.readOperands **do**
        if $v \notin b$.liveDef **then**
          b.liveUse.add(v)
        end if
      end for
      for virtual register v in i.writeOperands **do**
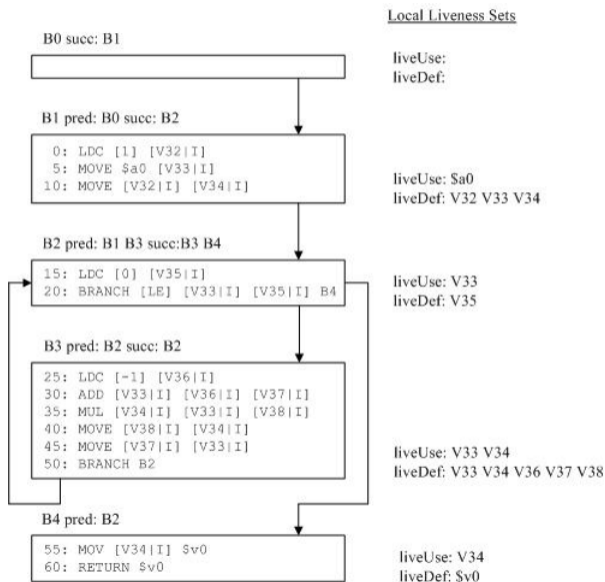 b.liveDef.add(v)
      end for
    end for
  end for

Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

The control-flow graph for `Factorial.computeIter()` with its local liveness sets computed is illustrated below

Local Liveness Sets

B0 succ: B1

liveUse:
liveDef:

B1 pred: B0 succ: B2

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

liveUse: $a0
liveDef: V32 V33 V34

B2 pred: B1 B3 succ:B3 B4

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

liveUse: V33
liveDef: V35

B3 pred: B2 succ: B2

```
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
```

liveUse: V33 V34
liveDef: V33 V34 V36 V37 V38

B4 pred: B2

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

liveUse: V34
liveDef: $v0

Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

We can compute the set of operands that are live at the beginning and end of a block using a backward data-flow analysis

Register Allocation by Graph Coloring

We can compute the set of operands that are live at the beginning and end of a block using a backward data-flow analysis

We call the set of operands that are live at the start of a block liveIn

Register Allocation by Graph Coloring

We can compute the set of operands that are live at the beginning and end of a block using a backward data-flow analysis

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

Register Allocation by Graph Coloring

We can compute the set of operands that are live at the beginning and end of a block using a backward data-flow analysis

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

---

Algorithm Computing Global Liveness Information

Input: The control-flow graph g for a method, and the local liveness sets liveUse and liveDef for every basic block

Output: Two sets for each basic block: liveIn, registers live at the beginning of the block, and liveOut, registers that are live at the end of the block

  repeat
    for block b in g.blocks in reverse order **do**
      b.liveOut $\leftarrow$ {}
      for block s in b.successors **do**
        b.liveOut $\leftarrow$ b.liveOut $\cup$ s.liveIn
      end for
      b.liveIn $\leftarrow$ (b.liveOut $-$ b.liveDef) $\cup$ b.liveUse
    end for
  until no liveOut has changed

Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

The computation of the global liveness information for `Factorial.computeIter()` requires three iterations

Register Allocation by Graph Coloring

The computation of the global liveness information for `Factorial.computeIter()` requires three iterations

After first iteration

```
B4 liveIn:   V34
B4 liveOut:
B3 liveIn:   V33 V34
B3 liveOut:
B2 liveIn:   V33 V34
B2 liveOut:  V33 V34
B1 liveIn:   $a0
B1 liveOut:  V33 V34
B0 liveIn:   $a0
B0 liveOut:  $a0
```

Register Allocation by Graph Coloring

The computation of the global liveness information for `Factorial.computeIter()` requires three iterations

After first iteration

```
B4 liveIn:  V34
B4 liveOut:
B3 liveIn:  V33 V34
B3 liveOut:
B2 liveIn:  V33 V34
B2 liveOut: V33 V34
B1 liveIn:  $a0
B1 liveOut: V33 V34
B0 liveIn:  $a0
B0 liveOut: $a0
```

After second iteration

```
B4 liveIn:  V34
B4 liveOut:
B3 liveIn:  V33 V34
B3 liveOut: V33 V34
B2 liveIn:  V33 V34
B2 liveOut: V33 V34
B1 liveIn:  $a0
B1 liveOut: V33 V34
B0 liveIn:  $a0
B0 liveOut: $a0
```

Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

After third (and final) iteration

```
B4 liveIn:  V34
B4 liveOut:
B3 liveIn:  V33 V34
B3 liveOut: V33 V34
B2 liveIn:  V33 V34
B2 liveOut: V33 V34
B1 liveIn:  $a0
B1 liveOut: V33 V34
B0 liveIn:  $a0
B0 liveOut: $a0
```

Register Allocation by Graph Coloring

After third (and final) iteration

```
B4 liveIn:  V34
B4 liveOut:
B3 liveIn:  V33 V34
B3 liveOut: V33 V34
B2 liveIn:  V33 V34
B2 liveOut: V33 V34
B1 liveIn:  $a0
B1 liveOut: V33 V34
B0 liveIn:  $a0
B0 liveOut: $a0
```

To build the intervals, we make a single pass over the blocks and instructions, again in reverse order

Algorithm Building Liveness Intervals

Input: The control-flow graph g for a method with LIR, and the liveIn and liveOut sets for each basic block
Output: A liveness interval for each register, with ranges and use positions

  for block b in g.blocks in reverse order **do**
    int $blockFrom \leftarrow b$.firstInstruction.id
    Set $blockTo \leftarrow b$.lastInstruction.id
    for register r in b.liveOut **do**
      intervals[r].addOrExtendRange($blockFrom$, $blockRange$)
    end for
    for instruction i in b.instructions in reverse order **do**
      if i.isAMethodCall **then**
        for physical register r in the set of physical registers **do**
          intervals[r].addRange(i.id, i.id)
        end for
      end if
      for virtual register r in i.writeOperands **do**
        intervals[r].firstRange.from $\leftarrow i$.id
        intervals[r].addUsePos(i.id)
      end for
      for virtual register r in i.readOperands **do**
        intervals[r].addOrExtendRange($blockFrom$, i.id)
        intervals[r].addUsePos(i.id)
      end for
    end for
  end for

Register Allocation by Graph Coloring

We first add ranges for all registers that are in liveOut, extending from the start of the block to the end

Register Allocation by Graph Coloring

We first add ranges for all registers that are in liveOut, extending from the start of the block to the end

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, because we assume the subroutine itself will use these registers and so we want to force spills
- 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, and a write may cause the start position to be re-adjusted — note that `addOrExtendRange()` operation merges contiguous ranges into one

Register Allocation by Graph Coloring

Consider the 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 the basic block is illustrated below
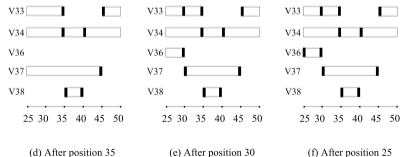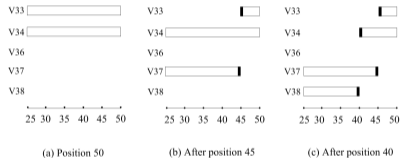
(a) Position 50   (b) After position 45   (c) After position 40

(d) After position 35   (e) After position 30   (f) After position 25

Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

In graph coloring register allocation, we start with an interference graph built from the liveness intervals

Register Allocation by Graph Coloring

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

Register Allocation by Graph Coloring

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, ie, if they are live at the same time

Register Allocation by Graph Coloring

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, ie, if they are live at the same time

For example, reconsider the liveness intervals for the virtual registers $V32 - $V38 of `Factorial.computeIter()`
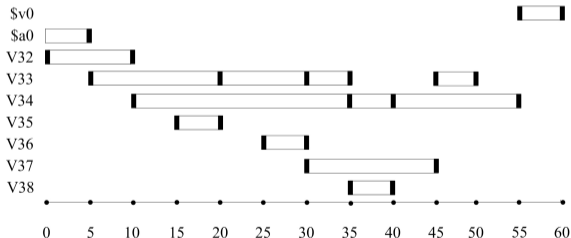
Register Allocation by Graph Coloring

The interference graph for `Factorial.computeIter()` is shown below

Register Allocation by Graph Coloring

The interference graph for `Factorial.computeIter()` is shown below

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

Register Allocation by Graph Coloring

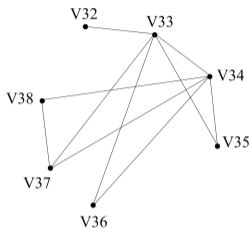The interference graph for `Factorial.computeIter()` is shown below
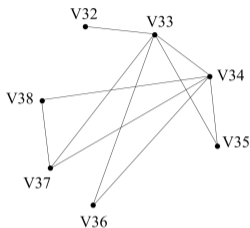
We say that a graph has an *R*-coloring if it can be colored using *R* distinct colors, or in our case *R* distinct physical registers

To exhaustively find such an *R*-coloring for $R \geq 2$ has long been known to be NP-complete

Register Allocation by Graph Coloring

The interference graph for `Factorial.computeIter()` is shown below

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
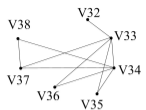
Register Allocation by Graph Coloring

① Degree $< R$ heuristic: a graph with a node of degree $< R$ 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; we continue removing nodes until either all nodes have been pruned or until we reach a state where all remaining nodes have degrees $\geq R$

② Optimistic heuristic: we use a function `spillCost()` to find a 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
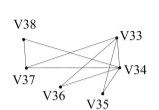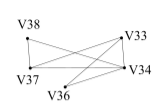
Register Allocation by Graph Coloring

Pruning of the above interference graph with $R = 3$ is shown below

(a)

(b) V32

(c) V35 V32

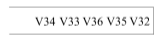(d) V36 V35 V32

(e) V33 V36 V35 V32

(f) V34 V33 V36 V35 V32

(g) V37 V34 V33 V36 V35 V32

(h) V38 V37 V34 V33 V36 V35 V32
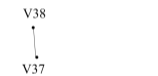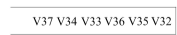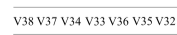
Register Allocation by Graph Coloring

Register Allocation by Graph Coloring

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 are never assigned the same physical register

Register Allocation by Graph Coloring

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 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

Register Allocation by Graph Coloring

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 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

```
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
```

Register Allocation by Graph Coloring

Algorithm Graph Coloring Register Allocation

Input: The control-flow graph g for a method with LIR that makes use of virtual registers
Output: The same g but with virtual registers replaced by physical registers

   registersAssignedSuccessfully ← false
  repeat
    repeat
      buildIntervals()
      buildInterferenceGraph()
    until not coalesceRegistersSuccessful()
    buildAdjacencyLists()
    computeSpillCosts()
    pruneGraph()
    registersAssignedSuccessfully ← assignRegisters()
    if not *registersAssignedSuccessfully* **then**
      generateSpillCode()
    end if
  until *registersAssignedSuccessfully*

Coalescing registers reduces both the number of virtual registers and the number of moves

Register Allocation by Graph Coloring

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 is used to insure that any register coalescing is followed by a rebuilding of the intervals and the interference graph

Register Allocation by Graph Coloring

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 is used to insure that any register coalescing is followed by a rebuilding of the intervals and the interference graph

We use both an adjacency matrix and an adjacency list representation for the interference graph

Register Allocation by Graph Coloring

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 is used to insure that any register coalescing is followed by a rebuilding of the intervals and the interference graph

We use both an adjacency matrix and an adjacency list representation for the interference graph

During the pruning process we may reach a state where only nodes with degree $\geq R$ remain in the graph, in which case, our algorithm must choose a node with the smallest spill cost

Register Allocation by Graph Coloring

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 is used to insure that any register coalescing is followed by a rebuilding of the intervals and the interference graph

We use both an adjacency matrix and an adjacency list representation for the interference graph

During the pruning process we may reach a state where only nodes with degree $\geq R$ remain in the graph, in which case, our algorithm must choose a node with the smallest spill cost

The spill cost of a register depends on the loop depths of the positions where the register must be stored to or loaded from memory

Register Allocation by Graph Coloring

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 is used to insure that any register coalescing is followed by a rebuilding of the intervals and the interference graph

We use both an adjacency matrix and an adjacency list representation for the interference graph

During the pruning process we may reach a state where only nodes with degree $\geq R$ remain in the graph, in which case, our algorithm must choose a node with the smallest spill cost

The spill cost of a register depends on the loop depths of the positions where the register must be stored to or loaded from memory

An option is summing up the uses and definitions, using a factor of 10^{depth} for taking loop depth into account; it's better to re-compute the value for a register when it is cheaper than spilling and reloading that same value