Compiler Construction

Chapter 13: Register allocation

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Introduction

- Registers are the fastest locations in the memory hierarchy.
- Most ALUs are working with registers.
- Use of registers is the critical factor in runtime performance.
- The register allocator determines, at each point in the program, which values will reside in the registers and which register will hold each of those values.
- The allocator might relegate a value to a memory
  - Because codes contain more live values than the number of registers
  - Because it is unsafe to store value in the register

Objectives

In a memory-to-memory model

- To keep values in the registers and eliminate some load/store instructions

In a register-to-register model

- To map virtual registers to physical registers and memory locations with added load/store instructions (spill codes)

To minimize the number of load and store instructions

Allocation vs. assignment

Allocation

- Maps unlimited space onto the set of the target machine
- Ensure that the code will fit the target machine's register set at each instruction
- NP-complete problem

Assignment

- Maps an allocated name set to the physical registers of the target machine
- Produce the actual register names
- Polynomial time

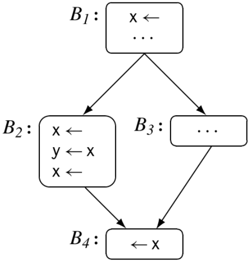
Register classes

- General-purpose registers
- Floating-point registers
- Predicate registers
- Branch-target registers

If the processor has no interactions between register classes, they can be allocated independently.

Live range

A closed set of related definitions and uses



Live Ranges in Code
with Control Flow

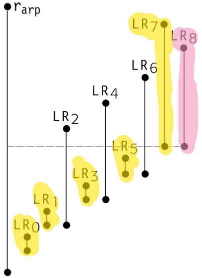
- SSA-forms, live until the exit point,etc.
- If the variable is ambiguous, it can only reside in a register between its creation and the next store operation in the code.

Live ranges in a basic block

	(a)		(b)	
1	loadAI	$r_{arp}, @a \Rightarrow r_a$	loadAI	$r_{arp}, @a \Rightarrow LR_7$
2	loadI	$2 \Rightarrow r_2$	loadI	$2 \Rightarrow LR_8$
3	loadAI	$r_{arp}, @b \Rightarrow r_b$	loadAI	$r_{arp}, @b \Rightarrow LR_6$
4	loadAI	$r_{arp}, @c \Rightarrow r_c$	loadAI	$r_{arp}, @c \Rightarrow LR_4$
5	loadAI	$r_{arp}, @d \Rightarrow r_d$	loadAI	$r_{arp}, @d \Rightarrow LR_2$
6	mult	$r_a, r_2 \Rightarrow r_a$	mult	$LR_7, LR_8 \Rightarrow LR_5$
7	mult	$r_a, r_b \Rightarrow r_a$	mult	$LR_5, LR_6 \Rightarrow LR_3$
8	mult	$r_a, r_c \Rightarrow r_a$	mult	$LR_3, LR_4 \Rightarrow LR_1$
9	mult	$r_a, r_d \Rightarrow r_a$	mult	$LR_1, LR_2 \Rightarrow LR_0$
10	storeAI	$r_a \Rightarrow r_{arp}, @a$	storeAI	$LR_0 \Rightarrow r_{arp}, @a$

(a) Example from Section 1.3.3

(b) Code Renamed into LRs



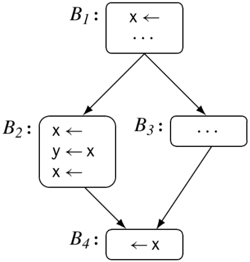
(c) Live Range Spans

FIGURE 13.1 Live Ranges in a Basic Block.

Interference

Two live ranges interfere if there exists an operation where both are live

- They should not share the same physical registers



Live Ranges in Code
with Control Flow

Spill

When the there is no available physical registers, the allocator must **spill** a range to memory, and **restore** it back before its subsequent use.

Spill cost

- Dirty value: need a store
- Clean value: no spill
- Rematerializable value: recompute is faster than load/store

Spill locations

- Usually, in `r_arp + offset`

Local register allocation and assignment

- Basic block scope
- The input block may contains more than k virtual registers, which is greater than the number of physical registers

Algorithm

1. Rename source registers into live ranges
2. Physical register allocation and assignment
3. Spill/restore

Data structure

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

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

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

■ **FIGURE 13.3** Representing a List of Operations.

Local register allocation

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

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

FIGURE 13.4 Renaming Source Registers into Live Ranges.

Local register allocation

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

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

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

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

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

    clear the mark in each PR           // reset marks

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

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

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

    return x

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

■ **FIGURE 13.5** The Local Allocator.

Spill and restore example: based on cost

			restore x_3
spill x_2		spill x_2	restore x_1
restore x_3	restore x_3	restore x_3	restore x_3
restore x_2	restore x_1	restore x_2	restore x_1
Spill Dirty	Spill Clean	Spill Dirty	Spill Clean

(a) References x_3 x_1 x_2

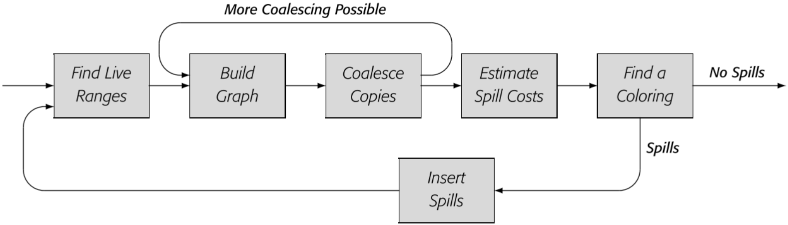
(b) References x_3 x_1 x_3 x_1 x_2

FIGURE 13.6 Spills of Clean Versus Dirty Values.

Global register alloc. and assignment

Simple steps

1. Find live ranges: might merge ranges from basic blocks
2. Build the (interference) graph
3. Coalesce copies: merge some ranges (nodes)
4. Estimate spill costs
5. Find a coloring
6. Insert spill



■ **FIGURE 13.7** Structure of the Global Coloring Allocator.

Graph coloring: interference graph

Interference graph: conflicts between live ranges

- Each node refer to a live range
- Each edge connects node (live ranges) which cannot share a register
- Color the interference graph: one color for one physical register
- Spilling will simplify the graph; hence reducing the number of colors required

We need to find the definition and its uses $\rightarrow$ SSA

- An operation that references the name defined by a ϕ-function uses the value of one of its arguments; which argument depends on how control flow reached the ϕ-function.
- All those definitions should reside in the same register and, thus, belong in the same live range.
- The algorithm examines each ϕ-function in the program, and unions together the (SSA) sets associated with each ϕ-function parameter and the set for the ϕ-function result

Example: Finding live ranges

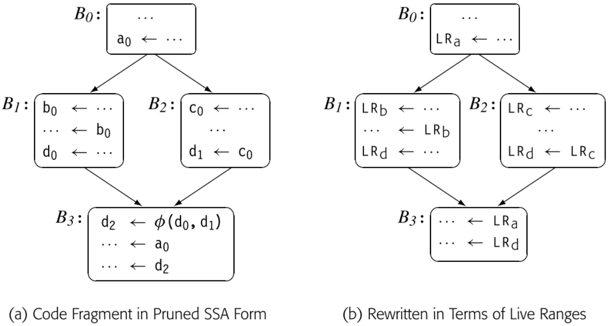


FIGURE 13.8 Discovering Live Ranges.

Estimating global spill costs

Cost of spilling

- Address computation: could be save if it is in the activation record
- Memory operation: load/store
- Estimated execution frequency: heuristic, data profile

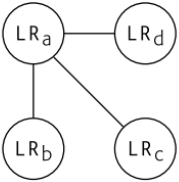
A live range can have

- Negative spill cost: spilling is cheaper than copy $\longrightarrow$ we should spill the range
- Infinite spill cost: spilling is useless $\longrightarrow$ we should not spill the range

Interference graph

Interference

- Two live ranges, LR_i and LR_j interfere if one is live at the definition of the other and they have different values.



An Interference Graph

Building the interference graph

Chula Chulalongkorn University

for each LR_i *do*

  create a node $n_i \in N$

for each basic block b do

  LIVENOW $\leftarrow$ LIVEOUT(*b*)

  for each operation i in b, from bottom to top,

    assuming form $op_i \;\; LR_a, LR_b \Rightarrow LR_c$ *do*

    remove LR_c *from* LIVENOW

    for each $LR_i \in$ LIVENOW† *do*

      add (LR_i, LR_c) *to* E

    add LR_a *and* LR_b *to* LIVENOW

† *If the operation is a copy,* $LR_i \Rightarrow LR_j$*, do not add the edge* (LR_i, LR_j).

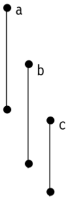
■ FIGURE 13.9 Constructing the Interference Graph.

Coalescing copies to reduce degree

Consider the operation $i2i LR_i \implies LR_j$

- If LR_i and LR_j do not otherwise interference
- We can rewrite all references to LR_j to LR_i

```
add LRt, LRu  ⇒ LRa
      ...
i2i LRa       ⇒ LRb
add LRa, LRb  ⇒ LRv
i2i LRa       ⇒ LRc
      ...
add LRb, LRw  ⇒ LRx
add LRc, LRy  ⇒ LRz
```



(a) Before Coalescing

```
add LRt, LRu    ⇒ LRab
      ...

add LRab, LRab  ⇒ LRv
i2i LRab        ⇒ LRc
      ...
add LRab, LRw   ⇒ LRx
add LRc, LRy    ⇒ LRz
```

(b) After Coalescing LR_a and LR_b

FIGURE 13.10 Combining Live Ranges by Coalescing Copy Operations.

Coalesce the most frequently executed copies first

Graph coloring

Finding K-coloring is NP-complete. Hence, we need an fast approximation of the algorithm.

- Simple graph coloring
- Spill nodes that do not have colors