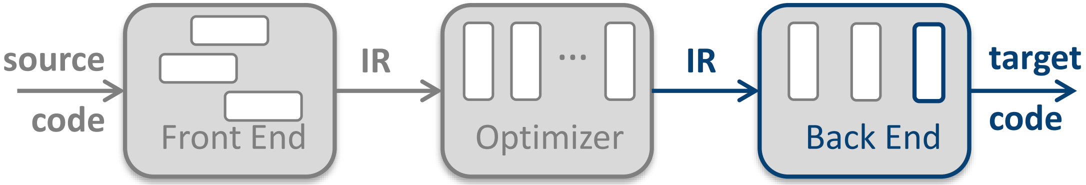
COMP 412
FALL 2018

Lab 2: Improving the Quality of Allocation

Comp 412



Lab 2 Schedule

Code checks 1 & 2 are milestones, not exhaustive tests. Many students still find bugs after code check 2, to say nothing of tuning for effectiveness & efficiency.

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
9	10	11	12	13	14 **Lab 2 Specs available**	15
16	17	18 **Tutorial** **5 PM** **McMurtry**	19	20	21 **Deadline:** **Code Check 1**	22
23	24 **Tutorial** **5 PM** **McMurtry**	25 **Dan Grove Talk** **(Dart Group)**	26	27	28	29
30	1 **Deadline:** **Code Check 2**	2 **Advice: Pay attention to the timing / scaling tests**	3	4	5	6
7	8 **Deadline:** **Lab 2 Code**	9	10	12 **Deadline:** **Lab Report**	12	13

Focus on connecting to Lab 1 IR, and Rename

Allocate correctly

Improve performance & allocation

YOU ARE HERE

Strategy

Lab 2 & Lab 3 are open-ended, in the sense that you can often improve performance by working on the lab for a while longer

- **Make managerial decisions about how to spend your time**
  - Get something working by the code check
    - → *Correctness is the goal, not a great allocation*
  - Tough decision at the early code due date
- **Take steps to protect against your own tinkering**
  - Checkpoint your code early and often
    - → *Each time something is working, checkpoint it*
    - → *Use GitHub if you aren't already*
  - Make notes with each checkpoint so that you will know what you have
    - → *Notes and comments protect you from memory loss & exhaustion*
    - → *Your commit messages should be informative*
- **Automate your testing**
  - Use one of the scripts that iterates over a directory of test codes

Bottom-up Allocator

A bottom-up allocator synthesizes the allocation based on low-level knowledge of the code & the partial solution thus far computed.

```
for i ← 0 to n
    if (OP[i].OP1.PR is invalid)
        get a PR, say x, load OP[i].OP1.VR into x, and set OP[i].OP1.PR ← x
    if (OP[i].OP2.PR is invalid)
        get a PR, say y, load OP[i].OP2.VR into y, and set OP[i].OP2.PR ← y
    if either OP1 or OP2 is a last use
        free the corresponding PR
    Get a PR, say z, and set OP[i].OP3.PR ← z
```

The action “get a **PR**” is the heart of the algorithm.

- If a **PR** is free, “get a PR” is easy
- If no **PR** is free, code must choose an occupied **PR**, spill its contents to memory, and record the location for use in a later restore operation

Bottom-up Allocator

“Get a PR”

- If some **PR**, say **prx**, is available, can use **prx** to hold the **VR** and either
  - **Restore** the value from memory (for a *use*), or
  - Use **prx** as the target (result) register (for a *def*)
- If all **PR**’s are in use, the bottom-up allocator must free one
  - Select the **PR** whose next use is farthest in the future, say **pry**
  - **Spill** the current contents of **pry** to memory (into its “spill location”)
  - Use **pry** to hold the **VR**, as discussed above
    - → For a *use*, **restore** the value into **pry**; for a *def*, **pry** becomes the target register

Save address of the “spill location” with the virtual register whose value it holds.

Complications will arise

- Different values have different costs to spill and restore. How does your allocator make that decision / tradeoff?
- Are ties an issue?
- Must your allocator reserve a spill register for the entire block, or can it use that register for data in a region where **MaxLive** ≤ *k* ?

Performance: What does it mean?

In Lab 2, big chunk of the points depend on the quality of the allocated code that your lab produces

- We measure performance by looking at the total number of cycles that the allocated code uses in the **Lab 2 ILOC Simulator**[1]
  - Take the unallocated code & run it
  - Take the allocated code & run it
  - Make sure the answers are identical
  - Difference between the two runs is the cost of spill code
- The objective of Lab 2 is to minimize the cost of the spill code
  - The credit for performance is based on how your lab does relative to the labs of other students (& relative to the reference implementation)
  - Register allocation can introduce a **lot** of spill code
  - Attention to detail can reduce spill costs
  - **But**, you cannot always win

[1] Lab 3 will use a different simulator, with different latencies & execution order constraints.

Allocation Quality

In local allocation, quality equates to the cost of the spills & restores

- **Spill** ⇒ preserving the value outside of a physical register
  - Store the value to memory, or
  - Recognize that the code can directly recreate the value
- **Restore** ⇒ recreating a spilled value in a physical register
  - Load the value from the location to which it was spilled, or
  - Recreate it directly, if possible

We call both of these "spill code."

How do we measure cost?

- Sum of the costs of the added operations
- In straight-line, or branch-free, code, the calculation is simple
  - Local allocation is the simple case

Categorizing Spills

Spills fall into three categories

General Case (*a "dirty" value; result of a computation*)

- Requires an address and a store to spill
- Requires an address and a load to restore
- *Total cost:* 8 cycles

The "reserved" register is used to hold the addresses for spills & restores.

Clean Value (*result of a load from memory or a previous spill*)

- Because value can be re-loaded, it requires no code to spill [1]
- Requires an address and a load to restore
- *Total cost:* 4 cycles

Rematerializable Value (*result of a loadI*)

- Because value can be recreated with loadI, it requires no code to spill
- Requires a single loadI to restore
- *Total cost:* 1 cycle

[1] Any store makes all previously "clean" values "dirty", unless the allocator can prove that the store defines a different memory location.

The General Case, ***k*** ≥ 3

The Bottom-up Local Algorithm makes spill decisions

- Heuristic says that when a register is needed, the allocator should spill the value whose next use (**NU**) is farthest in the future to free a register
  - This strategy produces good results
  - Optimal *when all choices have the same cost*

- So, why is it difficult?
  - Two values tie for maximum **NU**
    - → *Take cheaper spill cost of the two alternatives*
  - Is it ever better to take a smaller **NU**?
    - → Absolutely.
  - Can you tell when that case arises?
    - → Not in polynomial time, unless ***P*** = ***NP***

```
 •••
loadI     16          => vr0
add       vr0, vr2    => vr1
     Need to spill here
•••
          Use vr0
•••
          Use vr0
•••
add       vr9, vr1    => vr5
•••
```

Spill vr0 or vr1?

Spill and Restore: Tie Breaking

Ties in next-use distance can arise

- Two values with equal next-use distance
  - r12 and r13 have same next use
  - Assume neither is rematerializable
  - No obvious basis to choose
- If r13 is the result of a **load**, with a known address (result of a **loadI**) and r12 is not
  - Might be able to **restore** r13 with a **loadI** followed by a **load**, avoiding the **spill**
  - A value that can be reloaded from its original location is a **CLEAN** value
    - → There cannot be a store between the original load and the restore
    - → The store might change the value in memory
  - No telling how often this happens

```
add     r2,r8       =>   r12
loadI   132         =>   r10
load    r10         =>   13
  …
mult    r12, r13    =>   r25
sub     r17, r25    =>   r26
  …
add     r12, r13    =>   r32
  …
```

Need to spill here →

The Effect of Demand

The “best” spill choice depends heavily on context

0	loadI	…	
1	load	…	
2	add	…	
3	add	…	⇠ **spill here**
4	**mult**	**…**	
5	**store**	**…**	
6	**loadI**	**…**	
7	**rshift**	**…**	
8	store	…	
9	mult	…	
10	sub	…	⇠ **NU**(pr2)
11	add	…	
12	loadI	…	
13	load	…	
14	mult	…	⇠ **NU**(pr1)
15	store	…	

- **Boldface** indicates region where |LIVE| > ***k***
- At the spill, farthest use of the **PR**s is late in the block, say **pr1** in op 14
- If **pr1** is *dirty*, and **pr2** is *clean*, and **pr2** has its **NU** after the high-pressure region, say op 10, then spilling **pr2** before op 4 achieves the same goal at lower cost.
  - To make this decision requires perfect knowledge about defs, uses, and demand
- More complex scenarios arise
  - 2 *rematerializable* restores versus a *clean* one
  - 1 *clean* + 1 *rematerializable* versus 1 dirty
  - Multiple disjoint regions of high demand
- The problem is **NP Complete**

The Effect of Demand

The “best” spill choice depends heavily on context

0	loadI	…	
1	load	…	
2	add	…	
3	add	…	
4	**mult**	**…**	⇽ **spill here**
5	store	**…**	⇽ max remat
6	loadI	**…**	
7	rshift	**…**	
8	store	…	
9	mult	…	
10	sub	…	⇽ max clean
11	add	…	
12	loadI	…	
13	load	…	
14	mult	…	⇽ max dirty
15	store	…	

- If only one op has high pressure, taking *max remat* is obviously right
  - Minimizes cost

Pressure is defined as *demand for registers*, or **|LIVE|**.

The Effect of Demand

The “best” spill choice depends heavily on context

0	loadI	…	
1	load	…	
2	add	…	
3	add	…	**⇽ spill here**
4	**mult**	**…**	
5	**store**	**…**	⇽ max remat
6	**loadI**	**…**	
7	rshift	…	
8	store	…	
9	mult	…	
10	sub	…	⇽ max clean
11	add	…	
12	loadI	…	
13	load	…	
14	mult	…	⇽ max dirty
15	store	…	

- If only one op has high pressure, taking *max remat* is obviously right
  - Minimizes cost
- With a longer region of high pressure

The Effect of Demand

The “best” spill choice depends heavily on context

```
0   loadI   …
1   load    …
2   add     …
3   add     …    ← spill here
4   mult    …
5   store   …    ← max remat
6   loadI   …
7   rshift  …
8   store   …
9   mult    …
10  sub     …    ← max clean
11  add     …
12  loadI   …
13  load    …
14  mult    …    ← max dirty
15  store   …
```

- If only one op has high pressure, taking *max remat* is obviously right
  - Minimizes cost
- With a longer region of high pressure, or multiple regions of high pressure, *max clean* might be a better choice
  - Cover entire region with one spill
  - Tradeoff depends on low-level detail

The Effect of Demand

The “best” spill choice depends heavily on context

0	loadI	…	
1	load	…	
2	add	…	
3	add	…	⇠ **spill here**
4	**mult**	**…**	
5	store	…	⇠ max remat
6	loadI	…	
7	rshift	…	
8	store	…	
9	mult	…	
10	sub	…	⇠ max clean
11	add	…	
12	**loadI**	**…**	
13	**load**	**…**	
14	**mult**	**…**	⇠ max dirty
15	store	…	

- If only one op has high pressure, taking *max remat* is obviously right
  - Minimizes cost
- With a longer region of high pressure, or multiple regions of high pressure, *max clean* might be a better choice
  - Cover entire region with one spill
  - Tradeoff depends on low-level detail
- High pressure between *max clean* & *max dirty* might, or might not, make *max dirty* the right choice
  - What if result of op 6 is still around at op 12? *Clean* + *remat* might beat *dirty*.

The Effect of Demand

The "best" spill choice depends heavily on context

0	loadI	…	
1	load	…	
2	add	…	
3	add	…	
4	**mult**	**…**	⇜ **spill here**
5	store	…	⇜ max remat
6	loadI	…	
7	rshift	…	
8	store	…	
9	mult	…	
10	sub	…	⇜ max clean
11	add	…	
12	**loadI**	**…**	
13	**load**	**…**	
14	**mult**	**…**	⇜ max dirty
15	store	…	

- If only one op has high pressure, taking *max remat* is obviously right
  - Minimizes cost
- With a longer regi[illegible]e, or multip[illegible]
  - [illegible] one spill
  - [illegible]pends on low-level detail
- High pressure between *max clean* & *max dirty* might, or might not, make *max dirty* the right choice
  - What if result of op 6 is still around at op 12? Clean + remat might beat dirty.

The whole problem is NP Complete! You are unlikely to discover a heuristic that always produces the code that you want.

So, What Should A Local Allocator Do?

Carefully consider reasonable choices in a consistent strategy

- First, get the allocator working well with the default heuristic
  - Spill *max* ***NU***

So, What Should A Local Allocator Do?

Carefully consider reasonable choices in a consistent strategy

- First, get the allocator working well with the default heuristic
  - Spill *max* ***NU***
  - ***Checkpoint that allocator !***

So, What Should A Local Allocator Do?

Carefully consider reasonable choices in a consistent strategy

- First, get the allocator working well with the default heuristic
  - Spill *max* ***NU***
- When finding *max* **NU**, find the *max* for all three of rematerializable, clean, and dirty values
  - During your forward pass to allocate, keep track of the difference between these three on a **VR**-by-**VR** basis [1]
  - To find the *max*, the allocator will loop over the **PR**s; finding three values is only a little more work than finding one (*adds a small constant factor*)

---

[1] Remember that a store makes all clean values dirty.

So, What Should A Local Allocator Do?

Carefully consider reasonable choices in a consistent strategy

- First, get the allocator working well with the default heuristic
  - Spill *max* ***NU***
- When finding *max* **NU**, find the *max* for all three of rematerializable, clean, and dirty values
  - During your forward pass to allocate, keep track of the difference between these three on a **VR**-by-**VR** basis [1]
  - To find the *max*, the allocator will loop over the **PR**s; finding three values is only a little more work than finding one (*adds a small constant factor*)
- Develop a heuristic that gives you good results
  - Always pick the cheapest, or
  - Pick clean over dirty if the next uses are within *x* operations, or …
    - → *Situations like this one make compiler construction seem like both art & science*

---

[1] Remember that a store makes all clean values dirty.

So, What Should A Local Allocator Do?

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates

So, What Should A Local Allocator Do?

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates
- Checkpoint your allocator regularly as you change the heuristics
  - Checkpoint each working version
  - Keep them until after the code due date

So, What Should A Local Allocator Do?

Manage your development process for self-protection

- Test your heuristic widely
  - Report blocks, plus one of the timing blocks (all have the same pattern)
  - Libraries of contributed blocks range from trivial to hard
  - Invent new test blocks & share them with your classmates
- Checkpoint your allocator regularly as you change the heuristics
  - Checkpoint each working version
  - Keep them until after the code due date
- Post your questions to Piazza
  - The “community effect” is powerful

What About That Reserved Register?

Reserving a spill register has a large impact on performance

- *Especially when k is small*

- Your allocator only needs a reserved register when it needs to spill inside a region of high pressure (|**Live**| > *k*)
  - If spill is outside the region, spare registers are available to hold address
  - For a restore, allocator can use same register to hold address & value
- Once your allocator is working, you could teach it to only reserve a register in places where it absolutely needs one
  - Creates, on the margin, an extra register by "shrinking" the regions of high pressure
  - For a one-operation region of high-pressure, with |**Live**| = *k* + 1, where the region needs a restore but does not spill inside the region, you could avoid the reserved register

Subtleties with **CLEAN** Values

The interaction between CLEAN and stores requires some attention

- A value is considered **CLEAN** at some point in the code iff
  - It occupies a physical register at that point
  - Its value is guaranteed to exist in a known location in memory
- A **CLEAN** value can be **spilled** without a **store**, since it can be **restored** with a **loadI**, **load** pair from its known location

Observations

- Once the allocator spills and restores a value, that restored value is always **CLEAN**
  - A **store** in the original code cannot overwrite the spill address
- A value loaded from a known location in the block’s data memory (address ≤ 32764) is **CLEAN** until a store occurs
  - If the allocator knows the address of <u>both</u> the **load** and the **store**, it can determine if the store changes the location of the (formerly) **CLEAN** value

Subtleties with **CLEAN** Values

To track CLEAN values, you need to track stores

- During renaming, the allocator can build an ordered list of **store**s
- When the allocator needs to spill, it can compare the next use values against the line number of the next store to determine if the value is **CLEAN** or **DIRTY**
  - Adding this test creates a simple place to add more sophisticated tests, such as checking if both the **load** and **store** are based on *rematerializable* address and, if so, if those addresses are the same.
- This approach keeps to a linear-time complexity by pre-computing the necessary information.
  - (At each spill, check the line number of the store against the current line number and discard any stores that are "in the past.")

It is not at all clear how much improvement **CLEAN** values represent in practice. Get your allocator **working** well before you mess with this issue.

Special Case, *k* = 2

When *k* = 2, the allocator cannot reserve a register for the spill address

With just two registers, the compiler cannot, *in general*, keep a value in a register across the boundary between two operations

In general, the code will:

- Load the operand or operands before the operation
- Store the result after the operation

add vr1, vr2 => vr0 *becomes*

Schema for *k* = 2

*Get **vr1** into **pr0***

*Get **vr2** into **pr1***

add pr0, pr1 => pr1

Store pr1 to memory

I would use a specialized & simpler allocator for ***k*** = 2 because it cannot reserve a specific register for the spill address.

Special Case, *k* = 2

When *k* = 2, the allocator cannot reserve a register for the spill address

With just two registers, the compiler cannot keep a value in a register across the boundary between two operations

In general, the code will:

- Load the operand or operands before the operation
- Store the result after the operation

```
add     vr1, vr2     => vr0
```

becomes

Filling in the code

```
loadI     @vr1       => pr0
load      pr0        => pr0
loadI     @vr2       => pr1
load      pr1        => pr1
add       pr0, pr1   => pr1
loadI     @vr0       => pr0
store     pr1        => pr0
```

Restore vr1

Restore vr2

Spill vr1

(needs both registers)

The allocator should choose the lowest cost option for each restore & each spill.

Special Case, *k* = 2

Example

- Following the basic template for ***k*** = 2 leads to major code growth

… define & spill r2 …

```
loadI   17       => r1
add     r1, r2   => r3
add     r1, r3   => r4
```

no later use of r3

```
loadI   17        => pr1
loadI   @w        => pr0     Spill SR1 to
store   pr1       => pr0     @w
loadI   @w        => pr0     Restore
load    pr0       => pr0     SR1 to PR0
loadI   @x        => pr1     Restore
load    pr1       => pr1     SR2 to PR1
add     pr0, pr1  => pr1
loadI   @y        => pr0     Spill SR3 to
store   pr1       => pr0     @y
loadI   @w        => pr0     Restore
load    pr0       => pr0     SR1 to PR0
loadI   @y        => pr1     Restore
load    pr1       => pr1     SR3 to PR1
add     pr0, pr1  => pr1
loadI   @z        => pr0     Spill SR4 to
store   pr1       => pr0     @z
```

Here, @w, @x, @y, and @z are spill addresses generated by the allocator

3 ops became 17 ops.

Special Case, *k* = 2

Even at *k* = 2, there are some special cases

- With ***k*** = 2, the allocator must spill every result, unless:
  - *The result of operation i is not **LIVE** after operation i, or*
  - *The **only** use of operation i's result is in operation i+1*
  - *This reuse may free the allocator to preserve another common operand*

vrx has **NU** of *i+1*, and its **NU** in op *i+1* is ∞

- With ***k*** = 2, the allocator should still recognize ***rematerializable values*** and load them with a duplicate of the operation that created them
  - Do not spill the result of **loadI 17 => r1**
  - Instead, mark it as *rematerializable*, and restore it, when needed, with a copy of the original **loadI** operation

- Do not spill ***clean values***
  - Instead, restore them from the original location

(a little bookkeeping)

Special Case, *k* = 2

Example

… define & spill r2 …

```
loadI   17      => r1
add     r1, r2  => r3
add     r1, r3  => r4
```

no later use of r3

```
loadI   17        => pr0
loadI   @x        => pr1
load    pr1       => pr1
add     pr0, pr1  => pr1
add     pr0, pr1  => pr1
loadI   @z        => pr0
store   pr1       => pr0
```

Did not spill 17 & then re-load it

Did not spill **SR3** Kept **SR1** in **PR0**

7 ops rather than 17 ops

Choice of pr1 (rather than pr0) was critical to reuse of the value "17" in pr0.

Here, @x and @z are appropriate spill addresses for **SR2** and **SR4**

Slides on Handling $k = 2$ Start Here

Unused since 2014